Chapter 1

Introduction

1.1 Scope

IEEE Std 1003.1-200x defines a standard operating system interface and environment, including a command interpreter (or ‘‘shell’’), and common utility programs to support applications portability at the source code level. It is intended to be used by both applications developers and system implementors.

IEEE Std 1003.1-200x comprises four major components (each in an associated volume):

1. General terms, concepts, and interfaces common to all volumes of IEEE Std 1003.1-200x, including utility conventions and C language header definitions, are included in the Base Definitions volume of IEEE Std 1003.1-200x.
2. Definitions for system service functions and subroutines, language-specific system services for the C programming language, function issues, including portability, error handling, and error recovery, are included in the System Interfaces volume of IEEE Std 1003.1-200x.
3. Definitions for a standard source code-level interface to command interpretation services (a ‘‘shell’’) and common utility programs for application programs are included in the Shell and Utilities volume of IEEE Std 1003.1-200x.
4. Extended rationale that did not fit well into the rest of the document structure, containing historical information concerning the contents of IEEE Std 1003.1-200x and why features were included or discarded by the standard developers, is included in the Rationale (Informative) volume of IEEE Std 1003.1-200x.

The following areas are outside of the scope of IEEE Std 1003.1-200x:

- Graphics interfaces
- Database management system interfaces
- Record I/O considerations
- Object or binary code portability
- System configuration and resource availability

IEEE Std 1003.1-200x describes the external characteristics and facilities that are of importance to applications developers, rather than the internal construction techniques employed to achieve these capabilities. Special emphasis is placed on those functions and facilities that are needed in a wide variety of commercial applications.

The facilities provided in IEEE Std 1003.1-200x are drawn from the following base documents:

- IEEE Std 1003.1-1996 (POSIX-1) (incorporating IEEE Stds. 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995)
- The following amendments to the POSIX.1-1990 standard:
  - — IEEE P1003.1a draft standard (Additional System Services)
  - — IEEE Std 1003.1d-1999 (Additional Realtime Extensions)

  - — IEEE Std 1003.1g-2000 (Protocol-Independent Interfaces (PII))
  - — IEEE Std 1003.1j-2000 (Advanced Realtime Extensions)
  - — IEEE Std 1003.1q-2000 (Tracing)
- IEEE Std 1003.2-1992 (POSIX-2) (includes IEEE Std 1003.2a-1992)
- The following amendments to the ISO POSIX-2: 1993 standard:
  - — IEEE P1003.2b draft standard (Additional Utilities)
  - — IEEE Std 1003.2d-1994 (Batch Environment)
- Open Group Technical Standard, February 1997, System Interface Definitions, Issue 5 (XBD5) (ISBN: 1-85912-186-1, C605)
- Open Group Technical Standard, February 1997, Commands and Utilities, Issue 5 (XCU5) (ISBN: 1-85912-191-8, C604)
- Open Group Technical Standard, February 1997, System Interfaces and Headers, Issue 5 (XSH5) (in 2 Volumes) (ISBN: 1-85912-181-0, C606)

  Note: XBD5, XCU5, and XSH5 are collectively referred to as the *Base Specifications*.

- Open Group Technical Standard, January 2000, Networking Services, Issue 5.2 (XNS5.2) (ISBN: 1-85912-241-8, C808)
- ISO / IEC 9899: 1999, Programming Languages — C.

IEEE Std 1003.1-200x uses the *Base Specifications* as its organizational basis and adds the following additional functionality to them drawn from the base documents above:

- Normative text from the ISO POSIX-1: 1996 standard and the ISO POSIX-2: 1993 standard not included in the *Base Specifications*
- The amendments to the POSIX.1-1990 standard and the ISO POSIX-2: 1993 standard listed above, except for parts of IEEE Std 1003.1g-2000
- Portability Considerations
- Additional rationale and notes

The following features, marked legacy or obsolescent in the base documents, are not carried forward into IEEE Std 1003.1-200x. Other features from the base documents marked legacy or obsolescent are carried forward unless otherwise noted.

From XSH5, the following legacy interfaces, headers, and external variables are not carried forward:

> *advance*( ), *brk*( ), *chroot*( ), *compile*( ), *cuserid*( ), *gamma*( ), *getdtablesize*( ), *getpagesize*( ), *getpass*( ), *getw*( ), *putw*( ), *re_comp*( ), *re_exec*( ), *regcmp*( ), *sbrk*( ), *sigstack*( ), *step*( ), *wait3*( ), **<re_comp.h>**, **<regexp.h>**, **<varargs.h>**, *loc1*, *__loc1*, *loc2*, *locs*

From XCU5, the following legacy utilities are not carried forward:

> *calendar*, *cancel*, *cc*, *col*, *cpio*, *cu*, *dircmp*, *dis*, *egrep*, *fgrep*, *line*, *lint*, *lpstat*, *mail*, *pack*, *pcat*, *pg*, *spell*, *sum*, *tar*, *unpack*, *uulog*, *uuname*, *uupick*, *uuto*

In addition, legacy features within non-legacy reference pages (for example, headers) are not carried forward.

From the ISO POSIX-1: 1996 standard, the following obsolescent features are not carried forward:

Page 112, CLK_TCK
Page 197 *tcgetattr*( ) rate returned option

From the ISO POSIX-2: 1993 standard, obsolescent features within the following pages are not carried forward:

Page 75, zero-length prefix within PATH
Page 156, 159 *set*
Page 178, *awk*, use of no argument and no parentheses with length
Page 259, *ed*
Page 272, *env*
Page 282, *find* −**perm**[−]*onum*
Page 295-296, *egrep*
Page 299-300, *head*
Page 305-306, *join*
Page 309-310, *kill*
Page 431-433, 435-436, *sort*
Page 444-445, *tail*
Page 453, 455-456, *touch*
Page 464-465, *tty*
Page 472, *uniq*
Page 515-516, *ex*
Page 542-543, *expand*
Page 563-565, *more*
Page 574-576, *newgrp*
Page 578, *nice*
Page 594-596, *renice*
Page 597-598, *split*
Page 600-601, *strings*
Page 624-625, *vi*
Page 693, *lex*

The *c89* utility (which specified a compiler for the C Language specified by the ISO / IEC 9899: 1990 standard) has been replaced by a *c99* utility (which specifies a compiler for the C Language specified by the ISO / IEC 9899: 1999 standard).

From XSH5, text marked OH (Optional Header) has been reviewed on a case-by-case basis and removed where appropriate. The XCU5 text marked OF (Output Format Incompletely Specified) and UN (Possibly Unsupportable Feature) has been reviewed on a case-by-case basis and removed where appropriate |

For the networking interfaces, the base document is the XNS, Issue 5.2 specification. The following parts of the XNS, Issue 5.2 specification are out of scope and not included in IEEE Std 1003.1-200x:

- Part 3 (XTI)
- Part 4 (Appendixes)

Since there is much duplication between the XNS, Issue 5.2 specification and IEEE Std 1003.1g-2000, material only from the following sections of IEEE Std 1003.1g-2000 has been included:

- General terms related to sockets (2.2.2)
- Socket concepts (5.1 through 5.3, inclusive)

- The *pselect*( ) function (6.2.2.1 and 6.2.3)
- The **<sys/select.h>** header (6.2)

Emphasis is placed on standardizing existing practice for existing users, with changes and additions limited to correcting deficiencies in the following areas:

- Issues raised by IEEE or ISO / IEC Interpretations against IEEE Std 1003.1 and IEEE Std 1003.2
- Issues raised in corrigenda for the *Base Specifications* and working group resolutions from The Open Group
- Corrigenda and resolutions passed by The Open Group for the XNS, Issue 5.2 specification
- Changes to make the text self-consistent with the additional material merged
- A reorganization of the options in order to facilitate profiling, both for smaller profiles such as IEEE Std 1003.13, and larger profiles such as the Single UNIX Specification
- Alignment with the ISO / IEC 9899: 1999 standard

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-200x are defined in Chapter 2 (on page 15).

1.3 Normative References

The following standards contain provisions which, through references in IEEE Std 1003.1-200x, constitute provisions of IEEE Std 1003.1-200x. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on IEEE Std 1003.1-200x are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

ANS X3.9-1978

(Reaffirmed 1989) American National Standard for Information Systems: Standard X3.9-1978, Programming Language FORTRAN.[1]

ISO / IEC 646: 1991

ISO / IEC 646: 1991, Information Processing — ISO 7-bit Coded Character Set for Information Interchange.[2]

The reference version of the standard contains 95 graphic characters, which are identical to the graphic characters defined in the ASCII coded character set.

ISO 4217: 1995

ISO 4217: 1995, Codes for the Representation of Currencies and Funds. |

---

1. ANSI documents can be obtained from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018, U.S.A.
2. ISO/IEC documents can be obtained from the ISO office: 1 Rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland / Suisse

ISO 8601: 2000

ISO 8601: 2000, Data Elements and Interchange Formats — Information Interchange — Representation of Dates and Times.

ISO C (1999)

ISO / IEC 9899: 1999, Programming Languages — C, including Technical Corrigendum No. 1.

ISO / IEC 10646-1: 2000

ISO / IEC 10646-1: 2000, Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.

1.4 Terminology

For the purposes of IEEE Std 1003.1-200x, the following terminology definitions apply:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to IEEE Std 1003.1-200x. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by IEEE Std 1003.1-200x but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to IEEE Std 1003.1-200x. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

shall

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is mandatory. An application can rely on the existence of the feature or behavior.

For an application or user, describes a behavior that is mandatory.

should

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by IEEE Std 1003.1-200x which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by IEEE Std 1003.1-200x which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.5 Portability

Some of the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x and functions in the System Interfaces volume of IEEE Std 1003.1-200x describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance).

Where optional, enhanced, or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the option, extension, or warning (see Section 1.5.1). For maximum portability, an application should avoid such functionality.

Unless the primary task of a utility is to produce textual material on its standard output, application developers should not rely on the format or content of any such material that may be produced. Where the primary task *is* to provide such material, but the output format is incompletely specified, the description is marked with the OF margin code and shading. Application developers are warned not to expect that the output of such an interface on one system is any guide to its behavior on another system.

1.5.1 Codes

The codes and their meanings are as follows. See also Section 1.5.2 (on page 14).

ADV Advisory Information

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

AIO Asynchronous Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the AIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the AIO margin legend.

BAR **Barriers**

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the BAR margin legend.

BE **Batch Environment Services and Utilities**

The functionality described is optional.

Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the BE margin legend.

CD **C-Language Development Utilities**

The functionality described is optional.

Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the CD margin legend.

CPT **Process CPU-Time Clocks**

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CPT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CPT margin legend.

CS **Clock Selection**

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CS margin legend.

CX **Extension to the ISO C standard**

The functionality described is an extension to the ISO C standard. Application writers may make use of an extension as it is supported on all IEEE Std 1003.1-200x-conforming systems.

With each function or header from the ISO C standard, a statement to the effect that ‘‘any conflict is unintentional’’ is included. That is intended to refer to a direct conflict. IEEE Std 1003.1-200x acts in part as a profile of the ISO C standard, and it may choose to further constrain behaviors allowed to vary by the ISO C standard. Such limitations are not considered conflicts.

FD **FORTRAN Development Utilities**

The functionality described is optional.

Where applicable, utilities are marked with the FD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FD margin legend.

FR **FORTRAN Runtime Utilities**

The functionality described is optional.

Where applicable, utilities are marked with the FR margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FR margin legend.

FSC File Synchronization

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the FSC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the FSC margin legend.

IP6 IPV6

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the IP6 margin legend.

MC1 Advisory Information and either Memory Mapped Files or Shared Memory Objects

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC1 margin legend.

Refer to Section 1.5.2 (on page 14).

MC2 Memory Mapped Files, Shared Memory Objects, or Memory Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC2 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC2 margin legend.

Refer to Section 1.5.2 (on page 14).

MF Memory Mapped Files

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MF margin legend.

ML Process Memory Locking

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ML margin legend.

MLR

Range Memory Locking

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MLR margin legend.

MON

Monotonic Clock

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MON margin legend.

MPR

Memory Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MPR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MPR margin legend.

MSG

Message Passing

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MSG margin legend.

MX

IEC 60559 Floating-Point Option

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MX margin legend.

OB

Obsolescent

The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.

OF

Output Format Incompletely Specified

The functionality described is an XSI extension. The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OH

Optional Header

In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std 1003.1-200x an included header is marked as in the following example:

OH

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

This indicates that the marked header is not required on XSI-conformant systems.

PIO Prioritized Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.

PS Process Scheduling

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PS margin legend.

RS Raw Sockets

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RS margin legend.

RTS Realtime Signals Extension

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.

SD Software Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.

SEM Semaphores

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.

SHM Shared Memory Objects

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SHM margin legend.

SIO Synchronized Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SIO margin legend.

SPI Spin Locks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPI margin legend.

SPN Spawn

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPN margin legend.

SS Process Sporadic Server

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SS margin legend.

TCT Thread CPU-Time Clocks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TCT margin legend.

TEF Trace Event Filter

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TEF margin legend.

THR Threads

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the THR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the THR margin legend.

TMO Timeouts

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMO margin legend.

TMR

Timers

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMR margin legend.

TPI

Thread Priority Inheritance

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPI margin legend.

TPP

Thread Priority Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPP margin legend.

TPS

Thread Execution Scheduling

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPS margin legend.

TRC

Trace

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRC margin legend.

TRI

Trace Inherit

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRI margin legend.

TRL

Trace Log

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRL margin legend.

TSA

Thread Stack Address Attribute

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSA margin legend.

TSF Thread-Safe Functions

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSF margin legend.

TSH Thread Process-Shared Synchronization

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSH margin legend.

TSP Thread Sporadic Server

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSP margin legend.

TSS Thread Stack Address Size

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSS margin legend.

TYM Typed Memory Objects

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TYM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TYM margin legend.

UP User Portability Utilities

The functionality described is optional.

Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the UP margin legend.

XSI Extension

The functionality described is an XSI extension. Functionality marked XSI is also an extension to the ISO C standard. Application writers may confidently make use of an extension on all systems supporting the X/Open System Interfaces Extension.

If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that reference page is an extension. See Section 2.1.4 (on page 19).

XSR XSI STREAMS

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the XSR margin legend.

1.5.2 Margin Code Notation

Some of the functionality described in IEEE Std 1003.1-200x depends on support of more than one option, or independently may depend on several options. The following notation for margin codes is used to denote the following cases.

A Feature Dependent on One or Two Options

In this case, margin codes have a <space> separator; for example:

MF This feature requires support for only the Memory Mapped Files option.

MF SHM This feature requires support for both the Memory Mapped Files and the Shared Memory Objects options; that is, an application which uses this feature is portable only between implementations that provide both options.

A Feature Dependent on Either of the Options Denoted

In this case, margin codes have a '|' separator to denote the logical OR; for example:

MF|SHM This feature is dependent on support for either the Memory Mapped Files option or the Shared Memory Objects option; that is, an application which uses this feature is portable between implementations that provide any (or all) of the options.

A Feature Dependent on More than Two Options

The following shorthand notations are used:

MC1 The MC1 margin code is shorthand for ADV (MF|SHM). Features which are shaded with this margin code require support of the Advisory Information option and either the Memory Mapped Files or Shared Memory Objects option.

MC2 The MC2 margin code is shorthand for MF|SHM|MPR. Features which are shaded with this margin code require support of either the Memory Mapped Files, Shared Memory Objects, or Memory Protection options.

Large Sections Dependent on an Option

Where large sections of text are dependent on support for an option, a lead-in text block is provided and shaded accordingly; for example:

TRC This section describes extensions to support tracing of user applications. This functionality is dependent on support of the Trace option (and the rest of this section is not further shaded for this option).

Chapter 2

Conformance

2.1 Implementation Conformance

2.1.1 Requirements

A *conforming implementation* shall meet all of the following criteria:

1. The system shall support all utilities, functions, and facilities defined within IEEE Std 1003.1-200x that are required for POSIX conformance (see Section 2.1.3 (on page 16)). These interfaces shall support the functional behavior described herein.

2. The system may support one or more options as described under Section 2.1.5 (on page 20). When an implementation claims that an option is supported, all of its constituent parts shall be provided.

3. The system may support the X/Open System Interface Extension (XSI) as described under Section 2.1.4 (on page 19).

4. The system may provide additional utilities, functions, or facilities not required by IEEE Std 1003.1-200x. Non-standard extensions of the utilities, functions, or facilities specified in IEEE Std 1003.1-200x should be identified as such in the system documentation. Non-standard extensions, when used, may change the behavior of utilities, functions, or facilities defined by IEEE Std 1003.1-200x. The conformance document shall define an environment in which an application can be run with the behavior specified by IEEE Std 1003.1-200x. In no case shall such an environment require modification of a Strictly Conforming POSIX Application (see Section 2.2.1 (on page 29)).

2.1.2 Documentation

A conformance document with the following information shall be available for an implementation claiming conformance to IEEE Std 1003.1-200x. The conformance document shall have the same structure as IEEE Std 1003.1-200x, with the information presented in the appropriate sections and subsections. Sections and subsections that consist solely of subordinate section titles, with no other information, are not required. The conformance document shall not contain information about extended facilities or capabilities outside the scope of IEEE Std 1003.1-200x.

The conformance document shall contain a statement that indicates the full name, number, and date of the standard that applies. The conformance document may also list international software standards that are available for use by a Conforming POSIX Application. Applicable characteristics where documentation is required by one of these standards, or by standards of government bodies, may also be included.

The conformance document shall describe the limit values found in the headers **<limits.h>** (on page 245) and **<unistd.h>** (on page 398), stating values, the conditions under which those values may change, and the limits of such variations, if any.

The conformance document shall describe the behavior of the implementation for all implementation-defined features defined in IEEE Std 1003.1-200x. This requirement shall be met by listing these features and providing either a specific reference to the system documentation or providing full syntax and semantics of these features. When the value or behavior in the implementation is designed to be variable or customized on each instantiation of the system, the implementation provider shall document the nature and permissible ranges of this variation.

The conformance document may specify the behavior of the implementation for those features where IEEE Std 1003.1-200x states that implementations may vary or where features are identified as undefined or unspecified.

The conformance document shall not contain documentation other than that specified in the preceding paragraphs except where such documentation is specifically allowed or required by other provisions of IEEE Std 1003.1-200x.

The phrases ''shall document'' or ''shall be documented'' in IEEE Std 1003.1-200x mean that documentation of the feature shall appear in the conformance document, as described previously, unless there is an explicit reference in the conformance document to show where the information can be found in the system documentation.

The system documentation should also contain the information found in the conformance document.

2.1.3 POSIX Conformance

A conforming implementation shall meet the following criteria for POSIX conformance.

2.1.3.1 POSIX System Interfaces

- The system shall support all the mandatory functions and headers defined in IEEE Std 1003.1-200x, and shall set the symbolic constant _POSIX_VERSION to the value 200xxxL.
- Although all implementations conforming to IEEE Std 1003.1-200x support all the features described below, there may be system-dependent or file system-dependent configuration procedures that can remove or modify any or all of these features. Such configurations should not be made if strict compliance is required.

  The following symbolic constants shall either be undefined or defined with a value other than −1. If a constant is undefined, an application should use the *sysconf*( ), *pathconf*( ), or *fpathconf*( ) functions, or the *getconf* utility, to determine which features are present on the system at that time or for the particular pathname in question.

  — _POSIX_CHOWN_RESTRICTED

  The use of *chown*( ) is restricted to a process with appropriate privileges, and to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs.

  — _POSIX_NO_TRUNC

  Pathname components longer than {NAME_MAX} generate an error.

- The following symbolic constants shall be defined as follows:
  - _POSIX_JOB_CONTROL shall have a value greater than zero.
  - _POSIX_SAVED_IDS shall have a value greater than zero.
  - _POSIX_VDISABLE shall have a value other than −1.

  Note: The symbols above represent historical options that are no longer allowed as options, but are retained here for backwards-compatibility of applications.

- The system may support one or more options (see Section 2.1.6 (on page 26)) denoted by the following symbolic constants:
  - _POSIX_ADVISORY_INFO
  - _POSIX_ASYNCHRONOUS_IO
  - _POSIX_BARRIERS
  - _POSIX_CLOCK_SELECTION
  - _POSIX_CPUTIME
  - _POSIX_FSYNC
  - _POSIX_IPV6
  - _POSIX_MAPPED_FILES
  - _POSIX_MEMLOCK
  - _POSIX_MEMLOCK_RANGE
  - _POSIX_MEMORY_PROTECTION
  - _POSIX_MESSAGE_PASSING
  - _POSIX_MONOTONIC_CLOCK
  - _POSIX_PRIORITIZED_IO
  - _POSIX_PRIORITY_SCHEDULING
  - _POSIX_RAW_SOCKETS
  - _POSIX_REALTIME_SIGNALS
  - _POSIX_SEMAPHORES
  - _POSIX_SHARED_MEMORY_OBJECTS
  - _POSIX_SPAWN
  - _POSIX_SPIN_LOCKS
  - _POSIX_SPORADIC_SERVER
  - _POSIX_SYNCHRONIZED_IO
  - _POSIX_THREAD_ATTR_STACKADDR
  - _POSIX_THREAD_CPUTIME
  - _POSIX_THREAD_ATTR_STACKSIZE
  - _POSIX_THREAD_PRIO_INHERIT
  - _POSIX_THREAD_PRIO_PROTECT
  - _POSIX_THREAD_PRIORITY_SCHEDULING
  - _POSIX_THREAD_PROCESS_SHARED
  - _POSIX_THREAD_SAFE_FUNCTIONS
  - _POSIX_THREAD_SPORADIC_SERVER
  - _POSIX_THREADS

- — _POSIX_TIMEOUTS
- — _POSIX_TIMERS
- — _POSIX_TRACE
- — _POSIX_TRACE_EVENT_FILTER
- — _POSIX_TRACE_INHERIT
- — _POSIX_TRACE_LOG
- — _POSIX_TYPED_MEMORY_OBJECTS

If any of the symbolic constants _POSIX_TRACE_EVENT_FILTER, _POSIX_TRACE_LOG, or _POSIX_TRACE_INHERIT is defined to have a value other than –1, then the symbolic constant _POSIX_TRACE shall also be defined to have a value other than –1.

XSI

- The system may support the XSI extensions (see Section 2.1.5.2 (on page 21)) denoted by the following symbolic constants:
  - — _XOPEN_CRYPT
  - — _XOPEN_LEGACY
  - — _XOPEN_REALTIME
  - — _XOPEN_REALTIME_THREADS
  - — _XOPEN_UNIX

2.1.3.2 *POSIX Shell and Utilities*

- The system shall provide all the mandatory utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x with all the functional behavior described therein.
- The system shall support the Large File capabilities described in the Shell and Utilities volume of IEEE Std 1003.1-200x.
- The system may support one or more options (see Section 2.1.6 (on page 26)) denoted by the following symbolic constants. (The literal names below apply to the *getconf* utility.)
  - — POSIX2_C_DEV
  - — POSIX2_CHAR_TERM
  - — POSIX2_FORT_DEV
  - — POSIX2_FORT_RUN
  - — POSIX2_LOCALEDEF
  - — POSIX2_PBS
  - — POSIX2_PBS_ACCOUNTING
  - — POSIX2_PBS_LOCATE
  - — POSIX2_PBS_MESSAGE
  - — POSIX2_PBS_TRACK
  - — POSIX2_SW_DEV
  - — POSIX2_UPE

- The system may support the XSI extensions (see Section 2.1.4).

Additional language bindings and development utility options may be provided in other related standards or in a future version of IEEE Std 1003.1-200x. In the former case, additional symbolic constants of the same general form as shown in this subsection should be defined by the related standard document and made available to the application without requiring IEEE Std 1003.1-200x to be updated.

2.1.4 XSI Conformance

XSI This section describes the criteria for implementations conforming to the X/Open System Interface extension. This functionality is dependent on the support of the XSI extension (and the rest of this section is not further shaded).

IEEE Std 1003.1-200x describes utilities, functions, and facilities offered to application programs by the X/Open System Interface (XSI). An XSI-conforming implementation shall meet the criteria for POSIX conformance and the following requirements.

2.1.4.1 XSI System Interfaces

- The system shall support all the functions and headers defined in IEEE Std 1003.1-200x as part of the XSI extension denoted by the symbolic constant _XOPEN_UNIX and any extensions marked with the XSI extension marking (see Section 1.5.1 (on page 6)).
- The system shall support the *mmap*( ), *munmap*( ), and *msync*( ) functions.
- The system shall support the following options defined within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)):

  — _POSIX_FSYNC

  — _POSIX_MAPPED_FILES

  — _POSIX_MEMORY_PROTECTION

  — _POSIX_THREAD_ATTR_STACKADDR

  — _POSIX_THREAD_ATTR_STACKSIZE

  — _POSIX_THREAD_PROCESS_SHARED

  — _POSIX_THREAD_SAFE_FUNCTIONS

  — _POSIX_THREADS

- The system may support the following XSI Option Groups (see Section 2.1.5.2 (on page 21)) defined within IEEE Std 1003.1-200x: |

  — Encryption

  — Realtime

  — Advanced Realtime

  — Realtime Threads

  — Advanced Realtime Threads

  — Tracing

  — XSI STREAMS

  — Legacy

2.1.4.2 XSI Shell and Utilities Conformance

- The system shall support all the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x as part of the XSI extension denoted by the XSI marking in the SYNOPSIS section, and any extensions marked with the XSI extension marking (see Section 1.5.1 (on page 6)) within the text.
- The system shall support the User Portability Utilities option.
- The system shall support creation of locales (see Chapter 7 (on page 119)).
- The C-language Development utility *c99* shall be supported.
- The XSI Development Utilities option may be supported. It consists of the following software development utilities:

  admin
  cflow
  ctags
  cxref
  delta
  get
  m4
  prs
  rmdel
  sact
  sccs
  unget
  val
  what

- Within the utilities that are provided, functionality marked by the code OF (see Section 1.5.1 (on page 6)) need not be provided.

2.1.5 Option Groups

An Option Group is a group of related functions or options defined within the System Interfaces volume of IEEE Std 1003.1-200x.

If an implementation supports an Option Group, then the system shall support the functional behavior described herein.

If an implementation does not support an Option Group, then the system need not support the functional behavior described herein.

2.1.5.1 Subprofiling Considerations

Profiling standards supporting functional requirements less than that required in IEEE Std 1003.1-200x may subset both mandatory and optional functionality required for POSIX Conformance (see Section 2.1.3 (on page 16)) or XSI Conformance (see Section 2.1.4 (on page 19)). Such profiles shall organize the subsets into Subprofiling Option Groups.

The Rationale (Informative) volume of IEEE Std 1003.1-200x, Appendix E, Subprofiling Considerations (Informative) describes a representative set of such Subprofiling Option Groups for use by profiles applicable to specialized realtime systems. IEEE Std 1003.1-200x does not require that the presence of Subprofiling Option Groups be testable at compile-time (as symbols defined in any header) or at runtime (via *sysconf*( ) or *getconf*).

A Subprofiling Option Group may provide basic system functionality that other Subprofiling Option Groups and other options depend upon.[3] If a profile of IEEE Std 1003.1-200x does not

---

[3] As an example, the File System profiling option group provides underlying support for pathname resolution and file creation which are needed by any interface in IEEE Std 1003.1-200x that parses a *path* argument. If a profile requires support for the Device Input and Output profiling option group but does not require support for the File System profiling option group, the profile must specify how pathname resolution is to behave in that profile, how the O_CREAT flag to *open*( ) is to be handled (and the use of the character `'a'` in the *mode* argument of *fopen*( ) when a filename argument names a file that does not exist), and specify lots of other details.

require an implementation to provide a Subprofiling Option Group that provides features utilized by a required Subprofiling Option Group (or option),[4] the profile shall specify[5] all of the following:

- Restricted or altered behavior of interfaces defined in IEEE Std 1003.1-200x that may differ on an implementation of the profile
- Additional behaviors that may produce undefined or unspecified results
- Additional implementation-defined behavior that implementations shall be required to document in the profile's conformance document

if any of the above is a result of the profile not requiring an interface required by IEEE Std 1003.1-200x.

The following additional rules shall apply to all standard profiles of IEEE Std 1003.1-200x:

- Any application that conforms to that profile shall also conform to IEEE Std 1003.1-200x (that is, a profile cannot require restricted, altered, or extended behaviors).
- Any implementation that conforms to IEEE Std 1003.1-200x (including all options required by the profile) shall also conform to that profile

2.1.5.2 XSI Option Groups

XSI This section describes Option Groups to support the definition of XSI conformance within the System Interfaces volume of IEEE Std 1003.1-200x. This functionality is dependent on the support of the XSI extension (and the rest of this section is not further shaded).

The following Option Groups are defined.

Encryption

The Encryption Option Group is denoted by the symbolic constant _XOPEN_CRYPT. It includes the following functions:

crypt( ), *encrypt*( ), *setkey*( )

These functions are marked CRYPT.

Due to export restrictions on the decoding algorithm in some countries, implementations may be restricted in making these functions available. All the functions in the Encryption Option Group may therefore return [ENOSYS] or, alternatively, *encrypt*( ) shall return [ENOSYS] for the decryption operation.

An implementation that claims conformance to this Option Group shall set _XOPEN_CRYPT to a value other than −1.

---

[4] As an example, IEEE Std 1003.1-200x requires that implementations claiming to support the Range Memory Locking option also support the Process Memory Locking option. A profile could require that the Range Memory Locking option had to be supplied without requiring that the Process Memory Locking option be supplied as long as the profile specifies everything an application writer or system implementor would have to know to build an application or implementation conforming to the profile.

[5] Note that the profile could just specify that any use of the features not specified by the profile would produce undefined or unspecified results.

Realtime

The Realtime Option Group is denoted by the symbolic constant _XOPEN_REALTIME.

This Option Group includes a set of realtime functions drawn from options within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)).

Where entire functions are included in the Option Group, the NAME section is marked with REALTIME. Where additional semantics have been added to existing pages, the new material is identified by use of the appropriate margin legend for the underlying option defined within IEEE Std 1003.1-200x.

An implementation that claims conformance to this Option Group shall set _XOPEN_REALTIME to a value other than −1.

This Option Group consists of the set of the following options from within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)):

_POSIX_ASYNCHRONOUS_IO
_POSIX_FSYNC
_POSIX_MAPPED_FILES
_POSIX_MEMLOCK
_POSIX_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING
_POSIX_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING
_POSIX_REALTIME_SIGNALS
_POSIX_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SYNCHRONIZED_IO
_POSIX_TIMERS

If the symbolic constant _XOPEN_REALTIME is defined to have a value other than −1, then the following symbolic constants shall be defined by the implementation to have the value 200xxxL:

_POSIX_ASYNCHRONOUS_IO
_POSIX_MEMLOCK
_POSIX_MEMLOCK_RANGE
_POSIX_MESSAGE_PASSING
_POSIX_PRIORITY_SCHEDULING
_POSIX_REALTIME_SIGNALS
_POSIX_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SYNCHRONIZED_IO
_POSIX_TIMERS

The functionality associated with _POSIX_MAPPED_FILES, _POSIX_MEMORY_PROTECTION, and _POSIX_FSYNC is always supported on XSI-conformant systems.

Support of _POSIX_PRIORITIZED_IO on XSI-conformant systems is optional. If this functionality is supported, then _POSIX_PRIORITIZED_IO shall be set to a value other than −1. Otherwise, it shall be undefined.

If _POSIX_PRIORITIZED_IO is supported, then asynchronous I/O operations performed by *aio_read*( ), *aio_write*( ), and *lio_listio*( ) shall be submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*. The implementation shall also document for which files I/O prioritization is supported.

Advanced Realtime

An implementation that claims conformance to this Option Group shall also support the Realtime Option Group.

Where entire functions are included in the Option Group, the NAME section is marked with ADVANCED REALTIME. Where additional semantics have been added to existing pages, the new material is identified by use of the appropriate margin legend for the underlying option defined within IEEE Std 1003.1-200x.

This Option Group consists of the set of the following options from within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)):

_POSIX_ADVISORY_INFO
_POSIX_CLOCK_SELECTION
_POSIX_CPUTIME
_POSIX_MONOTONIC_CLOCK
_POSIX_SPAWN
_POSIX_SPORADIC_SERVER
_POSIX_TIMEOUTS
_POSIX_TYPED_MEMORY_OBJECTS

If the implementation supports the Advanced Realtime Option Group, then the following symbolic constants shall be defined by the implementation to have the value 200xxxL:

_POSIX_ADVISORY_INFO
_POSIX_CLOCK_SELECTION
_POSIX_CPUTIME
_POSIX_MONOTONIC_CLOCK
_POSIX_SPAWN
_POSIX_SPORADIC_SERVER
_POSIX_TIMEOUTS
_POSIX_TYPED_MEMORY_OBJECTS

If the symbolic constant _POSIX_SPORADIC_SERVER is defined, then the symbolic constant _POSIX_PRIORITY_SCHEDULING shall also be defined by the implementation to have the value 200xxxL.

If the symbolic constant _POSIX_CPUTIME is defined, then the symbolic constant _POSIX_TIMERS shall also be defined by the implementation to have the value 200xxxL.

If the symbolic constant _POSIX_MONOTONIC_CLOCK is defined, then the symbolic constant _POSIX_TIMERS shall also be defined by the implementation to have the value 200xxxL.

If the symbolic constant _POSIX_CLOCK_SELECTION is defined, then the symbolic constant _POSIX_TIMERS shall also be defined by the implementation to have the value 200xxxL.

Realtime Threads

The Realtime Threads Option Group is denoted by the symbolic constant _XOPEN_REALTIME_THREADS.

This Option Group consists of the set of the following options from within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)):

_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING

Where applicable, whole pages are marked REALTIME THREADS, together with the appropriate option margin legend for the SYNOPSIS section (see Section 1.5.1 (on page 6)).

An implementation that claims conformance to this Option Group shall set _XOPEN_REALTIME_THREADS to a value other than –1.

If the symbol _XOPEN_REALTIME_THREADS is defined to have a value other than –1, then the following options shall also be defined by the implementation to have the value 200xxxL:

_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING

Advanced Realtime Threads

An implementation that claims conformance to this Option Group shall also support the Realtime Threads Option Group.

Where entire functions are included in the Option Group, the NAME section is marked with ADVANCED REALTIME THREADS. Where additional semantics have been added to existing pages, the new material is identified by use of the appropriate margin legend for the underlying option defined within IEEE Std 1003.1-200x.

This Option Group consists of the set of the following options from within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)):

_POSIX_BARRIERS
_POSIX_SPIN_LOCKS
_POSIX_THREAD_CPUTIME
_POSIX_THREAD_SPORADIC_SERVER

If the symbolic constant _POSIX_THREAD_SPORADIC_SERVER is defined to have the value 200xxxL, then the symbolic constant _POSIX_THREAD_PRIORITY_SCHEDULING shall also be defined by the implementation to have the value 200xxxL.

If the symbolic constant _POSIX_THREAD_CPUTIME is defined to have the value 200xxxL, then the symbolic constant _POSIX_TIMERS shall also be defined by the implementation to have the value 200xxxL.

If the symbolic constant _POSIX_BARRIERS is defined to have the value 200xxxL, then the symbolic constants _POSIX_THREADS and _POSIX_THREAD_SAFE_FUNCTIONS shall also be defined by the implementation to have the value 200xxxL.

If the symbolic constant _POSIX_SPIN_LOCKS is defined to have the value 200xxxL, then the symbolic constants _POSIX_THREADS and _POSIX_THREAD_SAFE_FUNCTIONS shall also be defined by the implementation to have the value 200xxxL.

If the implementation supports the Advanced Realtime Threads Option Group, then the following symbolic constants shall be defined by the implementation to have the value 200xxxL:

_POSIX_BARRIERS
_POSIX_SPIN_LOCKS
_POSIX_THREAD_CPUTIME
_POSIX_THREAD_SPORADIC_SERVER

Tracing

This Option Group includes a set of tracing functions drawn from options within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)).

Where entire functions are included in the Option Group, the NAME section is marked with TRACING. Where additional semantics have been added to existing pages, the new material is identified by use of the appropriate margin legend for the underlying option defined within IEEE Std 1003.1-200x.

This Option Group consists of the set of the following options from within IEEE Std 1003.1-200x (see Section 2.1.6 (on page 26)):

_POSIX_TRACE
_POSIX_TRACE_EVENT_FILTER
_POSIX_TRACE_LOG
_POSIX_TRACE_INHERIT

If the implementation supports the Tracing Option Group, then the following symbolic constants shall be defined by the implementation to have the value 200xxxL:

_POSIX_TRACE
_POSIX_TRACE_EVENT_FILTER
_POSIX_TRACE_LOG
_POSIX_TRACE_INHERIT

XSI STREAMS

The XSI STREAMS Option Group is denoted by the symbolic constant _XOPEN_STREAMS.

This Option Group includes functionality related to STREAMS, a uniform mechanism for implementing networking services and other character-based I/O as described in the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.6, STREAMS.

It includes the following functions:

fattach( ), *fdetach*( ), *getmsg*( ), *getpmsg*( ), *ioctl*( ), *isastream*( ), *putmsg*( ), *putpmsg*( )

and the **<stropts.h>** header.

Where applicable, whole pages are marked STREAMS, together with the appropriate option margin legend for the SYNOPSIS section (see Section 1.5.1 (on page 6)). Where additional semantics have been added to existing pages, the new material is identified by use of the appropriate margin legend for the underlying option defined within IEEE Std 1003.1-200x.

An implementation that claims conformance to this Open Group shall set _XOPEN_STREAMS to a value other than –1.

Legacy

The Legacy Option Group is denoted by the symbolic constant _XOPEN_LEGACY.

The Legacy Option Group includes the functions and headers which were mandatory in previous versions of IEEE Std 1003.1-200x but are optional in this version.

These functions and headers are retained in IEEE Std 1003.1-200x because of their widespread use. Application writers should not rely on the existence of these functions or headers in new applications, but should follow the migration path detailed in the APPLICATION USAGE sections of the relevant pages.

Various factors may have contributed to the decision to mark a function or header LEGACY. In all cases, the specific reasons for the withdrawal of a function or header are documented on the relevant pages.

Once a function or header is marked LEGACY, no modifications are made to the specifications of such functions or headers other than to the APPLICATION USAGE sections of the relevant pages.

The functions and headers which form this Option Group are as follows:

bcmp( ), *bcopy*( ), *bzero*( ), *ecvt*( ), *fcvt*( ), *ftime*( ), *gcvt*( ), *getwd*( ), *index*( ), *mktemp*( ), *rindex*( ), *utimes*( ), *wcswcs*( )

An implementation that claims conformance to this Option Group shall set _XOPEN_LEGACY to a value other than −1.

2.1.6 Options

The symbolic constants defined in **<unistd.h>**, **Constants for Options and Option Groups** (on page 398) reflect implementation options for IEEE Std 1003.1-200x. These symbols can be used by the application to determine which optional facilities are present on the implementation. The *sysconf*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x or the *getconf* utility defined in the Shell and Utilities volume of IEEE Std 1003.1-200x can be used to retrieve the value of each symbol on each specific implementation to determine whether the option is supported.

Where an option is not supported, the associated utilities, functions, or facilities need not be present.

Margin codes are defined for each option (see Section 1.5.1 (on page 6)).

2.1.6.1 System Interfaces

Refer to **<unistd.h>**, **Constants for Options and Option Groups** (on page 398) for the list of options.

2.1.6.2 Shell and Utilities

Each of these symbols shall be considered valid names by the implementation. Refer to **<unistd.h>**, **Constants for Options and Option Groups** (on page 398).

The literal names shown below apply only to the *getconf* utility.

CD POSIX2_C_DEV

The system supports the C-Language Development Utilities option.

The utilities in the C-Language Development Utilities option are used for the development of C-language applications, including compilation or translation of C source code and complex program generators for simple lexical tasks and processing of context-free grammars.

The utilities listed below may be provided by a conforming system; however, any system claiming conformance to the C-Language Development Utilities option shall provide all of the utilities listed.

c99
lex
yacc

POSIX2_CHAR_TERM

The system supports the Terminal Characteristics option. This value need not be present on a system not supporting the User Portability Utilities option.

Where applicable, the dependency is noted within the description of the utility.

This option applies only to systems supporting the User Portability Utilities option. If supported, then the system supports at least one terminal type capable of all operations described in IEEE Std 1003.1-200x; see Section 10.2 (on page 181).

FD POSIX2_FORT_DEV

The system supports the FORTRAN Development Utilities option.

The *fort77* FORTRAN compiler is the only utility in the FORTRAN Development Utilities option. This is used for the development of FORTRAN language applications, including compilation or translation of FORTRAN source code.

The *fort77* utility may be provided by a conforming system; however, any system claiming conformance to the FORTRAN Development Utilities option shall provide the *fort77* utility.

FR POSIX2_FORT_RUN

The system supports the FORTRAN Runtime Utilities option.

The *asa* utility is the only utility in the FORTRAN Runtime Utilities option.

The *asa* utility may be provided by a conforming system; however, any system claiming conformance to the FORTRAN Runtime Utilities option shall provide the *asa* utility.

POSIX2_LOCALEDEF

The system supports the Locale Creation Utilities option.

If supported, the system supports the creation of locales as described in the *localedef* utility.

The *localedef* utility may be provided by a conforming system; however, any system claiming conformance to the Locale Creation Utilities option shall provide the *localedef* utility.

BE POSIX2_PBS

The system supports the Batch Environment Services and Utilities option (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Chapter 3, Batch Environment Services).

Note: The Batch Environment Services and Utilities option is a combination of mandatory and optional batch services and utilities. The POSIX_PBS symbolic constant implies the system supports all the mandatory batch services and utilities.

POSIX2_PBS_ACCOUNTING

The system supports the Batch Accounting option.

POSIX2_PBS_CHECKPOINT

The system supports the Batch Checkpoint/Restart option.

POSIX2_PBS_LOCATE

The system supports the Locate Batch Job Request option.

POSIX2_PBS_MESSAGE

The system supports the Batch Job Message Request option.

POSIX2_PBS_TRACK

The system supports the Track Batch Job Request option.

SD POSIX2_SW_DEV

The system supports the Software Development Utilities option.

The utilities in the Software Development Utilities option are used for the development of applications, including compilation or translation of source code, the creation and maintenance of library archives, and the maintenance of groups of inter-dependent programs.

The utilities listed below may be provided by the conforming system; however, any system claiming conformance to the Software Development Utilities option shall provide all of the utilities listed here.

ar
make
nm
strip

UP POSIX2_UPE

The system supports the User Portability Utilities option.

The utilities in the User Portability Utilities option shall be implemented on all systems that claim conformance to this option. Certain utilities are noted as having features that cannot be implemented on all terminal types; if the POSIX2_CHAR_TERM option is supported, the system shall support all such features on at least one terminal type; see Section 10.2 (on page 181).

Some of the utilities are required only on systems that also support the Software Development Utilities option, or the character-at-a-time terminal option (see Section 10.2 (on page 181)); such utilities have this noted in their DESCRIPTION sections. All of the other utilities listed are required only on systems that claim conformance to the User Portability Utilities option.

alias
at
batch
bg
crontab
split
ctags
df
du
ex
expand
fc
fg
file
jobs
man
mesg
more
newgrp
nice
nm
patch
ps
renice
split
strings
tabs
talk
time
tput
unalias
unexpand
uudecode
uuencode
vi
who
write

2.2 Application Conformance

All applications claiming conformance to IEEE Std 1003.1-200x shall use only language-dependent services for the C programming language described in Section 2.3 (on page 31), shall use only the utilities and facilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, and shall fall within one of the following categories.

2.2.1 Strictly Conforming POSIX Application

A Strictly Conforming POSIX Application is an application that requires only the facilities described in IEEE Std 1003.1-200x. Such an application:

1. Shall accept any implementation behavior that results from actions it takes in areas described in IEEE Std 1003.1-200x as *implementation-defined* or *unspecified*, or where IEEE Std 1003.1-200x indicates that implementations may vary
2. Shall not perform any actions that are described as producing *undefined* results
3. For symbolic constants, shall accept any value in the range permitted by IEEE Std 1003.1-200x, but shall not rely on any value in the range being greater than the minimums listed or being less than the maximums listed in IEEE Std 1003.1-200x
4. Shall not use facilities designated as *obsolescent*
5. Is required to tolerate and permitted to adapt to the presence or absence of optional facilities whose availability is indicated by Section 2.1.3 (on page 16)
6. For the C programming language, shall not produce any output dependent on any behavior described in the ISO/IEC 9899:1999 standard as *unspecified*, *undefined*, or *implementation-defined*, unless the System Interfaces volume of IEEE Std 1003.1-200x specifies the behavior
7. For the C programming language, shall not exceed any minimum implementation limit defined in the ISO/IEC 9899:1999 standard, unless the System Interfaces volume of IEEE Std 1003.1-200x specifies a higher minimum implementation limit
8. For the C programming language, shall define _POSIX_C_SOURCE to be 200xxxL before any header is included |

Within IEEE Std 1003.1-200x, any restrictions placed upon a Conforming POSIX Application shall restrict a Strictly Conforming POSIX Application.

2.2.2 Conforming POSIX Application

2.2.2.1 ISO/IEC Conforming POSIX Application

An ISO/IEC Conforming POSIX Application is an application that uses only the facilities described in IEEE Std 1003.1-200x and approved Conforming Language bindings for any ISO or IEC standard. Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other ISO or IEC standards used.

2.2.2.2 <National Body> Conforming POSIX Application

A <National Body> Conforming POSIX Application differs from an ISO/IEC Conforming POSIX Application in that it also may use specific standards of a single ISO/IEC member body referred to here as *<National Body>*. Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other <National Body> standards used.

2.2.3 Conforming POSIX Application Using Extensions

A Conforming POSIX Application Using Extensions is an application that differs from a Conforming POSIX Application only in that it uses non-standard facilities that are consistent with IEEE Std 1003.1-200x. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a Conforming POSIX Application. A Conforming POSIX Application Using Extensions shall be either an ISO/IEC Conforming POSIX Application Using Extensions or a <National Body> Conforming POSIX Application Using Extensions (see Section 2.2.2.1 (on page 29) and Section 2.2.2.2 (on page 29)).

2.2.4 Strictly Conforming XSI Application

A Strictly Conforming XSI Application is an application that requires only the facilities described in IEEE Std 1003.1-200x. Such an application:

1. Shall accept any implementation behavior that results from actions it takes in areas described in IEEE Std 1003.1-200x as *implementation-defined* or *unspecified*, or where IEEE Std 1003.1-200x indicates that implementations may vary
2. Shall not perform any actions that are described as producing *undefined* results
3. For symbolic constants, shall accept any value in the range permitted by IEEE Std 1003.1-200x, but shall not rely on any value in the range being greater than the minimums listed or being less than the maximums listed in IEEE Std 1003.1-200x |
4. Shall not use facilities designated as *obsolescent*
5. Is required to tolerate and permitted to adapt to the presence or absence of optional facilities whose availability is indicated by Section 2.1.4 (on page 19)
6. For the C programming language, shall not produce any output dependent on any behavior described in the ISO C standard as *unspecified*, *undefined*, or *implementation-defined*, unless the System Interfaces volume of IEEE Std 1003.1-200x specifies the behavior
7. For the C programming language, shall not exceed any minimum implementation limit defined in the ISO C standard, unless the System Interfaces volume of IEEE Std 1003.1-200x specifies a higher minimum implementation limit
8. For the C programming language, shall define _XOPEN_SOURCE to be 600 before any header is included

Within IEEE Std 1003.1-200x, any restrictions placed upon a Conforming POSIX Application shall restrict a Strictly Conforming XSI Application.

2.2.5 Conforming XSI Application Using Extensions

A Conforming XSI Application Using Extensions is an application that differs from a Strictly Conforming XSI Application only in that it uses non-standard facilities that are consistent with IEEE Std 1003.1-200x. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a Strictly Conforming XSI Application.

2.3 Language-Dependent Services for the C Programming Language

Implementors seeking to claim conformance using the ISO C standard shall claim POSIX conformance as described in Section 2.1.3 (on page 16).

2.4 Other Language-Related Specifications

IEEE Std 1003.1-200x is currently specified in terms of the shell command language and ISO C. Bindings to other programming languages are being developed.

If conformance to IEEE Std 1003.1-200x is claimed for implementation of any programming language, the implementation of that language shall support the use of external symbols distinct to at least 31 bytes in length in the source program text. (That is, identifiers that differ at or before the thirty-first byte shall be distinct.) If a national or international standard governing a language defines a maximum length that is less than this value, the language-defined maximum shall be supported. External symbols that differ only by case shall be distinct when the character set in use distinguishes uppercase and lowercase characters and the language permits (or requires) uppercase and lowercase characters to be distinct in external symbols.

Chapter 3

Definitions

For the purposes of IEEE Std 1003.1-200x, the terms and definitions given in Chapter 3 apply.

Note: No shading to denote extensions or options occurs in this chapter. Where the terms and definitions given in this chapter are used elsewhere in text related to extensions and options, they are shaded as appropriate.

3.1 Abortive Release

An abrupt termination of a network connection that may result in the loss of data.

3.2 Absolute Pathname

A pathname beginning with a single or more than two slashes; see also Section 3.266 (on page 69).

Note: Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.3 Access Mode

A particular form of access permitted to a file.

3.4 Additional File Access Control Mechanism

An implementation-defined mechanism that is layered upon the access control mechanisms defined here, but which do not grant permissions beyond those defined herein, although they may further restrict them.

Note: File Access Permissions are defined in detail in Section 4.4 (on page 95).

3.5 Address Space

The memory locations that can be referenced by a process or the threads of a process.

3.6 Advisory Information

An interface that advises the implementation on (portable) application behavior so that it can optimize the system.

3.7 Affirmative Response

An input string that matches one of the responses acceptable to the *LC_MESSAGES* category keyword **yesexpr**, matching an extended regular expression in the current locale.

Note: The *LC_MESSAGES* category is defined in detail in Section 7.3.6 (on page 148).

3.8 Alert

To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the <alert> to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case).

3.9 Alert Character (<alert>)

A character that in the output stream should cause a terminal to alert its user via a visual or audible notification. It is the character designated by `'\a'` in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

3.10 Alias Name

In the shell command language, a word consisting solely of underscores, digits, and alphabetics from the portable character set and any of the following characters: `'!'`, `'%'`, `','`, `'@'`.

Implementations may allow other characters within alias names as an extension.

Note: The portable character set is defined in detail in Section 6.1 (on page 111).

3.11 Alignment

A requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.

Note: See also the ISO C standard, Section B3.

3.12 Alternate File Access Control Mechanism

An implementation-defined mechanism that is independent of the access control mechanisms defined herein, and which if enabled on a file may either restrict or extend the permissions of a given user. IEEE Std 1003.1-200x defines when such mechanisms can be enabled and when they are disabled.

Note: File Access Permissions are defined in detail in Section 4.4 (on page 95).

3.13 Alternate Signal Stack

Memory associated with a thread, established upon request by the implementation for a thread, separate from the thread signal stack, in which signal handlers responding to signals sent to that thread may be executed.

3.14 Ancillary Data

Protocol-specific, local system-specific, or optional information. The information can be both local or end-to-end significant, header information, part of a data portion, protocol-specific, and implementation or system-specific.

3.15 Angle Brackets

The characters '<' (left-angle-bracket) and '>' (right-angle-bracket). When used in the phrase ‘‘enclosed in angle brackets’’, the symbol '<' immediately precedes the object to be enclosed, and '>' immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used.

3.16 Application

A computer program that performs some desired function.

3.17 Application Address

Endpoint address of a specific application.

3.18 Application Program Interface (API)

The definition of syntax and semantics for providing computer system services.

3.19 Appropriate Privileges

An implementation-defined means of associating privileges with a process with regard to the function calls, function call options, and the commands that need special privileges. There may be zero or more such means. These means (or lack thereof) are described in the conformance document.

Note: Function calls are defined in the System Interfaces volume of IEEE Std 1003.1-200x, and commands are defined in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.20 Argument

In the shell command language, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name.

Note: The Utility Argument Syntax is defined in detail in Section 12.1 (on page 197) and the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1.1, Command Search and Execution.

In the C language, an expression in a function call expression or a sequence of preprocessing tokens in a function-like macro invocation.

3.21 Arm (a Timer)

To start a timer measuring the passage of time, enabling notifying a process when the specified time or time interval has passed.

3.22 Asterisk

The character `'*'`.

3.23 Async-Cancel-Safe Function

A function that may be safely invoked by an application while the asynchronous form of cancelation is enabled. No function is async-cancel-safe unless explicitly described as such.

3.24 Asynchronous Events

Events that occur independently of the execution of the application.

3.25 Asynchronous Input and Output

A functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion.

3.26 Async-Signal-Safe Function

A function that may be invoked, without restriction, from signal-catching functions. No function is async-signal-safe unless explicitly described as such.

3.27 Asynchronously-Generated Signal

A signal that is not attributable to a specific thread. Examples are signals sent via *kill*( ), signals sent from the keyboard, and signals delivered to process groups. Being asynchronous is a property of how the signal was generated and not a property of the signal number. All signals may be generated asynchronously.

Note: The *kill*( ) function is defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.28 Asynchronous I/O Operation

An I/O operation that does not of itself cause the thread requesting the I/O to be blocked from further use of the processor.

This implies that the process and the I/O operation may be running concurrently.

3.29 Asynchronous I/O Completion

For an asynchronous read or write operation, when a corresponding synchronous read or write would have completed and when any associated status fields have been updated.

3.30 Authentication

The process of validating a user or process to verify that the user or process is not a counterfeit.

3.31 Authorization

The process of verifying that a user or process has permission to use a resource in the manner requested.

To ensure security, the user or process would also need to be authenticated before granting access.

3.32 Background Job

See *Background Process Group* in Section 3.34.

3.33 Background Process

A process that is a member of a background process group.

3.34 Background Process Group (or Background Job)

Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal.

3.35 Backquote

The character '`', also known as a *grave accent*.

3.36 Backslash

The character '\', also known as a *reverse solidus*.

3.37 Backspace Character (<backspace>)

A character that, in the output stream, should cause printing (or displaying) to occur one column position previous to the position about to be printed. If the position about to be printed is at the beginning of the current line, the behavior is unspecified. It is the character designated by '\b' in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the backspace function. The <backspace> defined here is not necessarily the ERASE special character.

Note: Special Characters are defined in detail in Section 11.1.9 (on page 187).

3.38 Barrier

A synchronization object that allows multiple threads to synchronize at a particular point in their execution.

3.39 Base Character

One of the set of characters defined in the Latin alphabet. In Western European languages other than English, these characters are commonly used with diacritical marks (accents, cedilla, and so on) to extend the range of characters in an alphabet.

3.40 Basename

The final, or only, filename in a pathname.

3.41 Basic Regular Expression (BRE)

A regular expression (see Section 3.316 (on page 76)) used by the majority of utilities that select strings from a set of character strings.

Note: Basic Regular Expressions are described in detail in Section 9.3 (on page 167).

3.42 Batch Access List

A list of user IDs and group IDs of those users and groups authorized to place batch jobs in a batch queue.

A batch access list is associated with a batch queue. A batch server uses the batch access list of a batch queue as one of the criteria in deciding to put a batch job in a batch queue.

3.43 Batch Administrator

A user that is authorized to modify all the attributes of queues and jobs and to change the status of a batch server.

3.44 Batch Client

A computational entity that utilizes batch services by making requests of batch servers.

Batch clients often provide the means by which users access batch services, although a batch server may act as a batch client by virtue of making requests of another batch server.

3.45 Batch Destination

The batch server in a batch system to which a batch job should be sent for processing.

Acceptance of a batch job at a batch destination is the responsibility of a receiving batch server. A batch destination may consist of a batch server-specific portion, a network-wide portion, or both. The batch server-specific portion is referred to as the *batch queue*. The network-wide portion is referred to as a *batch server name*.

3.46 Batch Destination Identifier

A string that identifies a specific batch destination.

A string of characters in the portable character set used to specify a particular batch destination.

Note: The portable character set is defined in detail in Section 6.1 (on page 111).

3.47 Batch Directive

A line from a file that is interpreted by the batch server. The line is usually in the form of a comment and is an additional means of passing options to the *qsub* utility.

Note: The *qsub* utility is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.48 Batch Job

A set of computational tasks for a computing system.

Batch jobs are managed by batch servers.

Once created, a batch job may be executing or pending execution. A batch job that is executing has an associated session leader (a process) that initiates and monitors the computational tasks of the batch job.

3.49 Batch Job Attribute

A named data type whose value affects the processing of a batch job.

The values of the attributes of a batch job affect the processing of that job by the batch server that manages the batch job.

3.50 Batch Job Identifier

A unique name for a batch job. A name that is unique among all other batch job identifiers in a batch system and that identifies the batch server to which the batch job was originally submitted.

3.51 Batch Job Name

A label that is an attribute of a batch job. The batch job name is not necessarily unique.

3.52 Batch Job Owner

The *username@hostname* of the user submitting the batch job, where *username* is a user name (see also Section 3.426 (on page 91)) and *hostname* is a network host name.

3.53 Batch Job Priority

A value specified by the user that may be used by an implementation to determine the order in which batch jobs are selected to be executed. Job priority has a numeric value in the range −1 024 to 1 023.

Note: The batch job priority is not the execution priority (nice value) of the batch job.

3.54 Batch Job State

An attribute of a batch job which determines the types of requests that the batch server that manages the batch job can accept for the batch job. Valid states include QUEUED, RUNNING, HELD, WAITING, EXITING, and TRANSITING.

3.55 Batch Name Service

A service that assigns batch names that are unique within the batch name space, and that can translate a unique batch name into the location of the named batch entity.

3.56 Batch Name Space

The environment within which a batch name is known to be unique.

3.57 Batch Node

A host containing part or all of a batch system.

A batch node is a host meeting at least one of the following conditions:

- Capable of executing a batch client
- Contains a routing batch queue
- Contains an execution batch queue

3.58 Batch Operator

A user that is authorized to modify some, but not all, of the attributes of jobs and queues, and may change the status of the batch server.

3.59 Batch Queue

A manageable object that represents a set of batch jobs and is managed by a single batch server.

Note: A set of batch jobs is called a batch queue largely for historical reasons. Jobs are selected from the batch queue for execution based on attributes such as priority, resource requirements, and hold conditions.

See also the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 3.1.2, Batch Queues.

3.60 Batch Queue Attribute

A named data type whose value affects the processing of all batch jobs that are members of the batch queue.

A batch queue has attributes that affect the processing of batch jobs that are members of the batch queue.

3.61 Batch Queue Position

The place, relative to other jobs in the batch queue, occupied by a particular job in a batch queue. This is defined in part by submission time and priority; see also Section 3.62.

3.62 Batch Queue Priority

The maximum job priority allowed for any batch job in a given batch queue.

The batch queue priority is set and may be changed by users with appropriate privilege. The priority is bounded in an implementation-defined manner.

3.63 Batch Rerunability

An attribute of a batch job indicating that it may be rerun after an abnormal termination from the beginning without affecting the validity of the results.

3.64 Batch Restart

The action of resuming the processing of a batch job from the point of the last checkpoint. Typically, this is done if the batch job has been interrupted because of a system failure.

3.65 Batch Server

A computational entity that provides batch services.

3.66 Batch Server Name

A string of characters in the portable character set used to specify a particular server in a network.

Note: The portable character set is defined in detail in Section 6.1 (on page 111).

3.67 Batch Service

Computational and organizational services performed by a batch system on behalf of batch jobs.

Batch services are of two types: *requested* and *deferred*.

Note: Batch Services are listed in the Shell and Utilities volume of IEEE Std 1003.1-200x, Table 3-5, Batch Services Summary.

3.68 Batch Service Request

A solicitation of services from a batch client to a batch server.

A batch service request may entail the exchange of any number of messages between the batch client and the batch server.

When naming specific types of service requests, the term request is qualified by the type of request, as in *Queue Batch Job Request* and *Delete Batch Job Request*.

3.69 Batch Submission

The process by which a batch client requests that a batch server create a batch job via a *Queue Job Request* to perform a specified computational task.

3.70 Batch System

A collection of one or more batch servers.

3.71 Batch Target User

The name of a user on the batch destination batch server.

The target user is the user name under whose account the batch job is to execute on the destination batch server.

3.72 Batch User

A user who is authorized to make use of batch services.

3.73 Bind

The process of assigning a network address to an endpoint.

3.74 Blank Character (<blank>)

One of the characters that belong to the **blank** character class as defined via the *LC_CTYPE* category in the current locale. In the POSIX locale, a <blank> is either a <tab> or a <space>.

3.75 Blank Line

A line consisting solely of zero or more <blank>s terminated by a <newline>; see also Section 3.144 (on page 52).

3.76 Blocked Process (or Thread)

A process (or thread) that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution.

3.77 Blocking

A property of an open file description that causes function calls associated with it to wait for the requested action to be performed before returning.

3.78 Block-Mode Terminal

A terminal device operating in a mode incapable of the character-at-a-time input and output operations described by some of the standard utilities.

Note: Output Devices and Terminal Types are defined in detail in Section 10.2 (on page 181).

3.79 Block Special File

A file that refers to a device. A block special file is normally distinguished from a character special file by providing access to the device in a manner such that the hardware characteristics of the device are not visible.

3.80 Braces

The characters `'{'` (left brace) and `'}'` (right brace), also known as *curly braces*. When used in the phrase ‘‘enclosed in (curly) braces’’ the symbol `'{'` immediately precedes the object to be enclosed, and `'}'` immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used.

3.81 Brackets

The characters `'['` (left-bracket) and `']'` (right-bracket), also known as *square brackets*. When used in the phrase ‘‘enclosed in (square) brackets’’ the symbol `'['` immediately precedes the object to be enclosed, and `']'` immediately follows it. When describing these characters in the portable character set, the names <left-square-bracket> and <right-square-bracket> are used.

3.82 Broadcast

The transfer of data from one endpoint to several endpoints, as described in RFC 919 and RFC 922.

3.83 Built-In Utility (or Built-In)

A utility implemented within a shell. The utilities referred to as *special built-ins* have special qualities. Unless qualified, the term built-in includes the special built-in utilities. *Regular built-ins* are not required to be actually built into the shell on the implementation, but they do have special command-search qualities.

Note: Special Built-In Utilities are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.14, Special Built-In Utilities.

Regular Built-In Utilities are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1.1, Command Search and Execution.

3.84 Byte

An individually addressable unit of data storage that is exactly an octet, used to store a character or a portion of a character; see also Section 3.87. A byte is composed of a contiguous sequence of 8 bits. The least significant bit is called the *low-order* bit; the most significant is called the *high-order* bit.

Note: The definition of byte from the ISO C standard is broader than the above and might accommodate hardware architectures with different sized addressable units than octets.

3.85 Byte Input/Output Functions

The functions that perform byte-oriented input from streams or byte-oriented output to streams: *fgetc*( ), *fgets*( ), *fprintf*( ), *fputc*( ), *fputs*( ), *fread*( ), *fscanf*( ), *fwrite*( ), *getc*( ), *getchar*( ), *gets*( ), *printf*( ), *putc*( ), *putchar*( ), *puts*( ), *scanf*( ), *ungetc*( ), *vfprintf*( ), and *vprintf*( ).

Note: Functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.86 Carriage-Return Character (<carriage-return>)

A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the <carriage-return> occurred. It is the character designated by `'\r'` in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line.

3.87 Character

A sequence of one or more bytes representing a single graphic symbol or control code.

Note: This term corresponds to the ISO C standard term multi-byte character, where a single-byte character is a special case of a multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed.

See the definition of the portable character set in Section 6.1 (on page 111) for a further explanation of the graphical representations of (abstract) characters, as opposed to character encodings.

3.88 Character Array

An array of elements of type **char**.

3.89 Character Class

A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the *LC_CTYPE* category in the current locale.

Note: The *LC_CTYPE* category is defined in detail in Section 7.3.1 (on page 122).

3.90 Character Set

A finite set of different characters used for the representation, organization, or control of data.

3.91 Character Special File

A file that refers to a device. One specific type of character special file is a terminal device file.

Note: The General Terminal Interface is defined in detail in Chapter 11 (on page 183).

3.92 Character String

A contiguous sequence of characters terminated by and including the first null byte.

3.93 Child Process

A new process created (by *fork*( ) or *spawn*( )) by a given process. A child process remains the child of the creating process as long as both processes continue to exist.

Note: The *fork*( ) and *spawn*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.94 Circumflex

The character `'^'`.

3.95 Clock

A software or hardware object that can be used to measure the apparent or actual passage of time.

The current value of the time measured by a clock can be queried and, possibly, set to a value within the legal range of the clock.

3.96 Clock Jump

The difference between two successive distinct values of a clock, as observed from the application via one of the ‘‘get time’’ operations.

3.97 Clock Tick

An interval of time; an implementation-defined number of these occur each second. Clock ticks are one of the units that may be used to express a value found in type **clock_t**.

3.98 Coded Character Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation.

3.99 Codeset

The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned.

Note: Character Encoding is defined in detail in Section 6.2 (on page 114).

3.100 Collating Element

The smallest entity used to determine the logical ordering of character or wide-character strings; see also Section 3.102. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the *LC_COLLATE* category in the current locale determines the current set of collating elements.

3.101 Collation

The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements.

3.102 Collation Sequence

The relative order of collating elements as determined by the setting of the *LC_COLLATE* category in the current locale. The collation sequence is used for sorting and is determined from the collating weights assigned to each collating element. In the absence of weights, the collation sequence is the order in which collating elements are specified between **order_start** and **order_end** keywords in the *LC_COLLATE* category.

Multi-level sorting is accomplished by assigning elements one or more collation weights, up to the limit {COLL_WEIGHTS_MAX}. On each level, elements may be given the same weight (at the primary level, called an equivalence class; see also Section 3.150 (on page 53)) or be omitted from the sequence. Strings that collate equally using the first assigned weight (primary ordering) are then compared using the next assigned weight (secondary ordering), and so on.

Note: {COLL_WEIGHTS_MAX} is defined in detail in **<limits.h>**.

3.103 Column Position

A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in IEEE Std 1003.1-200x, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1.

3.104 Command

A directive to the shell to perform a particular task.

Note: Shell Commands are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9, Shell Commands.

3.105 Command Language Interpreter

An interface that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. It is possible for applications to invoke utilities through a number of interfaces, which are collectively considered to act as command interpreters. The most obvious of these are the *sh* utility and the *system*( ) function, although *popen*( ) and the various forms of *exec* may also be considered to behave as interpreters.

Note: The *sh* utility is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x.

The *system*( ), *popen*( ), and *exec* functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.106 Composite Graphic Symbol

A graphic symbol consisting of a combination of two or more other graphic symbols in a single character position, such as a diacritical mark and a base character.

3.107 Condition Variable

A synchronization object which allows a thread to suspend execution, repeatedly, until some associated predicate becomes true. A thread whose execution is suspended on a condition variable is said to be blocked on the condition variable.

3.108 Connection

An association established between two or more endpoints for the transfer of data

3.109 Connection Mode

The transfer of data in the context of a connection; see also Section 3.110.

3.110 Connectionless Mode

The transfer of data other than in the context of a connection; see also Section 3.109 and Section 3.123 (on page 49).

3.111 Control Character

A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text.

3.112 Control Operator

In the shell command language, a token that performs a control function. It is one of the following symbols:

```
&   &&   (   )   ;   ;;   newline   |   ||
```

The end-of-input indicator used internally by the shell is also considered a control operator.

Note: Token Recognition is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.3, Token Recognition.

3.113 Controlling Process

The session leader that established the connection to the controlling terminal. If the terminal subsequently ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process.

3.114 Controlling Terminal

A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal.

Note: The General Terminal Interface is defined in detail in Chapter 11 (on page 183).

3.115 Conversion Descriptor

A per-process unique value used to identify an open codeset conversion.

3.116 Core File

A file of unspecified format that may be generated when a process terminates abnormally.

3.117 CPU Time (Execution Time)

The time spent executing a process or thread, including the time spent executing system services on behalf of that process or thread. If the Threads option is supported, then the value of the CPU-time clock for a process is implementation-defined. With this definition the sum of all the execution times of all the threads in a process might not equal the process execution time, even in a single-threaded process, because implementations may differ in how they account for time during context switches or for other reasons.

3.118 CPU-Time Clock

A clock that measures the execution time of a particular process or thread.

3.119 CPU-Time Timer

A timer attached to a CPU-time clock.

3.120 Current Job

In the context of job control, the job that will be used as the default for the *fg* or *bg* utilities. There is at most one current job; see also Section 3.203 (on page 60).

3.121 Current Working Directory

See *Working Directory* in Section 3.436 (on page 92).

3.122 Cursor Position

The line and column position on the screen denoted by the terminal's cursor.

3.123 Datagram

A unit of data transferred from one endpoint to another in connectionless mode service.

3.124 Data Segment

Memory associated with a process, that can contain dynamically allocated data.

3.125 Deferred Batch Service

A service that is performed as a result of events that are asynchronous with respect to requests.

Note: Once a batch job has been created, it is subject to deferred services.

3.126 Device

A computer peripheral or an object that appears to the application as such.

3.127 Device ID

A non-negative integer used to identify a device.

3.128 Directory

A file that contains directory entries. No two directory entries in the same directory have the same name.

3.129 Directory Entry (or Link)

An object that associates a filename with a file. Several directory entries can associate names with the same file.

3.130 Directory Stream

A sequence of all the directory entries in a particular directory. An open directory stream may be implemented using a file descriptor.

3.131 Disarm (a Timer)

To stop a timer from measuring the passage of time, disabling any future process notifications (until the timer is armed again).

3.132 Display

To output to the user's terminal. If the output is not directed to a terminal, the results are undefined.

3.133 Display Line

A line of text on a physical device or an emulation thereof. Such a line will have a maximum number of characters which can be presented.

Note: This may also be written as ‘‘line on the display’’.

3.134 Dollar Sign

The character `'$'`.

3.135 Dot

In the context of naming files, the filename consisting of a single dot character (`'.'`).

Note: In the context of shell special built-in utilities, see *dot* in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.14, Special Built-In Utilities.

Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.136 Dot-Dot

The filename consisting solely of two dot characters (`".."`).

Note: Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.137 Double-Quote

The character `'"'`, also known as *quotation-mark*.

Note: The *double* adjective in this term refers to the two strokes in the character glyph. IEEE Std 1003.1-200x never uses the term double-quote to refer to two apostrophes or quotation marks.

3.138 Downshifting

The conversion of an uppercase character that has a single-character lowercase representation into this lowercase representation.

3.139 Driver

A module that controls data transferred to and received from devices.

Note: Drivers are traditionally written to be a part of the system implementation, although they are frequently written separately from the writing of the implementation. A driver may contain processor-specific code, and therefore be non-portable.

3.140 Effective Group ID

An attribute of a process that is used in determining various permissions, including file access permissions; see also Section 3.188 (on page 58).

3.141 Effective User ID

An attribute of a process that is used in determining various permissions, including file access permissions; see also Section 3.425 (on page 91).

3.142 Eight-Bit Transparency

The ability of a software component to process 8-bit characters without modifying or utilizing any part of the character in a way that is inconsistent with the rules of the current coded character set.

3.143 Empty Directory

A directory that contains, at most, directory entries for dot and dot-dot, and has exactly one link to it, in dot-dot. No other links to the directory may exist. It is unspecified whether an implementation can ever consider the root directory to be empty.

3.144 Empty Line

A line consisting of only a <newline>; see also Section 3.75 (on page 43).

3.145 Empty String (or Null String)

A string whose first byte is a null byte.

3.146 Empty Wide-Character String

A wide-character string whose first element is a null wide-character code.

3.147 Encoding Rule

The rules used to convert between wide-character codes and multi-byte character codes.

Note: Stream Orientation and Encoding Rules are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.5.2, Stream Orientation and Encoding Rules.

3.148 Entire Regular Expression

The concatenated set of one or more basic regular expressions or extended regular expressions that make up the pattern specified for string selection.

Note: Regular Expressions are defined in detail in Chapter 9 (on page 165).

3.149 Epoch

The time zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time (UTC).

Note: See also Seconds Since the Epoch defined in Section 4.14 (on page 100).

3.150 Equivalence Class

A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight.

3.151 Era

A locale-specific method for counting and displaying years.

Note: The *LC_TIME* category is defined in detail in Section 7.3.5 (on page 142).

3.152 Event Management

The mechanism that enables applications to register for and be made aware of external events such as data becoming available for reading.

3.153 Executable File

A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file.

3.154 Execute

To perform command search and execution actions, as defined in the Shell and Utilities volume of IEEE Std 1003.1-200x; see also Section 3.200 (on page 60).

Note: Command Search and Execution is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1.1, Command Search and Execution.

3.155 Execution Time

See *CPU Time* in Section 3.117 (on page 49).

3.156 Execution Time Monitoring

A set of execution time monitoring primitives that allow online measuring of thread and process execution times.

3.157 Expand

In the shell command language, when not qualified, the act of applying word expansions.

Note: Word Expansions are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions.

3.158 Extended Regular Expression (ERE)

A regular expression (see also Section 3.316 (on page 76)) that is an alternative to the Basic Regular Expression using a more extensive syntax, occasionally used by some utilities.

Note: Extended Regular Expressions are described in detail in Section 9.4 (on page 171).

3.159 Extended Security Controls

Implementation-defined security controls allowed by the file access permission and appropriate privilege (see also Section 3.19 (on page 35)) mechanisms, through which an implementation can support different security policies from those described in IEEE Std 1003.1-200x.

Note: See also Extended Security Controls defined in Section 4.3 (on page 95).

File Access Permissions are defined in detail in Section 4.4 (on page 95).

3.160 Feature Test Macro

A macro used to determine whether a particular set of features is included from a header.

Note: See also the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment.

3.161 Field

In the shell command language, a unit of text that is the result of parameter expansion, arithmetic expansion, command substitution, or field splitting. During command processing, the resulting fields are used as the command name and its arguments.

Note: Parameter Expansion is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.2, Parameter Expansion.

Arithmetic Expansion is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.4, Arithmetic Expansion.

Command Substitution is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3, Command Substitution.

Field Splitting is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.5, Field Splitting.

For further information on command processing, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1, Simple Commands.

3.162 FIFO Special File (or FIFO)

A type of file with the property that data written to such a file is read on a first-in-first-out basis.

Note: Other characteristics of FIFOs are described in the System Interfaces volume of IEEE Std 1003.1-200x, *lseek*( ), *open*( ), *read*( ), and *write*( ).

3.163 File

An object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, symbolic link, socket, and directory. Other types of files may be supported by the implementation.

3.164 File Description

See *Open File Description* in Section 3.253 (on page 67).

3.165 File Descriptor

A per-process unique, non-negative integer used to identify an open file for the purpose of file access. The value of a file descriptor is from zero to {OPEN_MAX}. A process can have no more than {OPEN_MAX} file descriptors open simultaneously. File descriptors may also be used to implement message catalog descriptors and directory streams; see also Section 3.253 (on page 67).

Note: {OPEN_MAX} is defined in detail in **<limits.h>**.

3.166 File Group Class

The property of a file indicating access permissions for a process related to the group identification of a process. A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file. Other members of the class may be implementation-defined.

3.167 File Mode

An object containing the *file mode bits* and file type of a file.

Note: File mode bits and file types are defined in detail in **<sys/stat.h>**.

3.168 File Mode Bits

A file's file permission bits, set-user-ID-on-execution bit (S_ISUID), and set-group-ID-on-execution bit (S_ISGID).

Note: File Mode Bits are defined in detail in **<sys/stat.h>**.

3.169 Filename

A name consisting of 1 to {NAME_MAX} bytes used to name a file. The characters composing the name may be selected from the set of all character values excluding the slash character and the null byte. The filenames dot and dot-dot have special meaning. A filename is sometimes referred to as a *pathname component*.

Note: Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.170 Filename Portability

Filenames should be constructed from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts. (For example, the use of a colon (':') in a pathname could cause ambiguity if that pathname were included in a *PATH* definition.)

3.171 File Offset

The byte position in the file where the next I/O operation begins. Each open file description associated with a regular file, block special file, or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a pipe or FIFO.

3.172 File Other Class

The property of a file indicating access permissions for a process related to the user and group identification of a process. A process is in the file other class of a file if the process is not in the file owner class or file group class.

3.173 File Owner Class

The property of a file indicating access permissions for a process related to the user identification of a process. A process is in the file owner class of a file if the effective user ID of the process matches the user ID of the file.

3.174 File Permission Bits

Information about a file that is used, along with other information, to determine whether a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes. These bits are contained in the file mode.

Note: File modes are defined in detail in **<sys/stat.h>**.

File Access Permissions are defined in detail in Section 4.4 (on page 95).

3.175 File Serial Number

A per-file system unique identifier for a file.

3.176 File System

A collection of files and certain of their attributes. It provides a name space for file serial numbers referring to those files.

3.177 File Type

See *File* in Section 3.163 (on page 55).

3.178 Filter

A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream.

3.179 First Open (of a File)

When a process opens a file that is not currently an open file within any process.

3.180 Flow Control

The mechanism employed by a communications provider that constrains a sending entity to wait until the receiving entities can safely receive additional data without loss.

3.181 Foreground Job

See *Foreground Process Group* in Section 3.183.

3.182 Foreground Process

A process that is a member of a foreground process group.

3.183 Foreground Process Group (or Foreground Job)

A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has at most one process group of the session as the foreground process group of that controlling terminal.

Note: The General Terminal Interface is defined in detail in Chapter 11.

3.184 Foreground Process Group ID

The process group ID of the foreground process group.

3.185 Form-Feed Character (<form-feed>)

A character that in the output stream indicates that printing should start on the next page of an output device. It is the character designated by `'\f'` in the C language. If the <form-feed> is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page.

3.186 Graphic Character

A member of the **graph** character class of the current locale.

Note: The **graph** character class is defined in detail in Section 7.3.1 (on page 122).

3.187 Group Database

A system database of implementation-defined format that contains at least the following information for each group ID:

- Group name
- Numerical group ID
- List of users allowed in the group

The list of users allowed in the group is used by the *newgrp* utility.

Note: The *newgrp* utility is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.188 Group ID

A non-negative integer, which can be contained in an object of type **gid_t**, that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs, or a saved set-group-ID.

3.189 Group Name

A string that is used to identify a group; see also Section 3.187. To be portable across conforming systems, the value is composed of characters from the portable filename character set. The hyphen should not be used as the first character of a portable group name.

3.190 Hard Limit

A system resource limitation that may be reset to a lesser or greater limit by a privileged process. A non-privileged process is restricted to only lowering its hard limit.

3.191 Hard Link

The relationship between two directory entries that represent the same file; see also Section 3.129 (on page 50). The result of an execution of the *ln* utility (without the −**s** option) or the *link*( ) function. This term is contrasted against symbolic link; see also Section 3.372 (on page 83).

3.192 Home Directory

The directory specified by the *HOME* environment variable.

3.193 Host Byte Order

The arrangement of bytes in any **int** type when using a specific machine architecture.

Note: Two common methods of byte ordering are big-endian and little-endian. Big-endian is a format for storage of binary data in which the most significant byte is placed first, with the rest in descending order. Little-endian is a format for storage or transmission of binary data in which the least significant byte is placed first, with the rest in ascending order. See also Section 4.8 (on page 97).

3.194 Incomplete Line

A sequence of one or more non-<newline>s at the end of the file.

3.195 Inf

A value representing +infinity or a value representing –infinity that can be stored in a floating type. Not all systems support the Inf values.

3.196 Instrumented Application

An application that contains at least one call to the trace point function *posix_trace_event*( ). Each process of an instrumented application has a mapping of trace event names to trace event type identifiers. This mapping is used by the trace stream that is created for that process.

3.197 Interactive Shell

A processing mode of the shell that is suitable for direct user interaction.

3.198 Internationalization

The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs, and coded character sets.

3.199 Interprocess Communication

A functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication.

3.200 Invoke

To perform command search and execution actions, except that searching for shell functions and special built-in utilities is suppressed; see also Section 3.154 (on page 53).

Note: Command Search and Execution is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1.1, Command Search and Execution.

3.201 Job

A set of processes, comprising a shell pipeline, and any processes descended from it, that are all in the same process group.

Note: See also the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.2, Pipelines.

3.202 Job Control

A facility that allows users selectively to stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter.

3.203 Job Control Job ID

A handle that is used to refer to a job. The job control job ID can be any of the forms shown in the following table:

Table 3-1 Job Control Job ID Formats

Job Control Job ID	Meaning
%%	Current job.
%+	Current job.
%–	Previous job.
%*n*	Job number *n*.
%*string*	Job whose command begins with *string*.
%?*string*	Job whose command contains *string*.

3.204 Last Close (of a File)

When a process closes a file, resulting in the file not being an open file within any process.

3.205 Line

A sequence of zero or more non-<newline>s plus a terminating <newline>.

3.206 Linger

A period of time before terminating a connection, to allow outstanding data to be transferred.

3.207 Link

See *Directory Entry* in Section 3.129 (on page 50).

3.208 Link Count

The number of directory entries that refer to a particular file.

3.209 Local Customs

The conventions of a geographical area or territory for such things as date, time, and currency formats.

3.210 Local Interprocess Communication (Local IPC)

The transfer of data between processes in the same system. |

3.211 Locale |

The definition of the subset of a user's environment that depends on language and cultural conventions. |

Note: Locales are defined in detail in Chapter 7 (on page 119).

3.212 Localization

The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets.

3.213 Login

The unspecified activity by which a user gains access to the system. Each login is associated with exactly one login name.

3.214 Login Name

A user name that is associated with a login.

3.215 Map

To create an association between a page-aligned range of the address space of a process and some memory object, such that a reference to an address in that range of the address space results in a reference to the associated memory object. The mapped memory object is not necessarily memory-resident.

3.216 Marked Message

A STREAMs message on which a certain flag is set. Marking a message gives the application protocol-specific information. An application can use *ioctl*( ) to determine whether a given message is marked.

Note: The *ioctl*( ) function is defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.217 Matched

A state applying to a sequence of zero or more characters when the characters in the sequence correspond to a sequence of characters defined by a basic regular expression or extended regular expression pattern.

Note: Regular Expressions are defined in detail in Chapter 9 (on page 165).

3.218 Memory Mapped Files

A facility to allow applications to access files as part of the address space.

3.219 Memory Object

One of:

- A file (see Section 3.163 (on page 55))
- A shared memory object (see Section 3.340 (on page 79))
- A typed memory object (see Section 3.418 (on page 90))

When used in conjunction with *mmap*( ), a memory object appears in the address space of the calling process.

Note: The *mmap*( ) function is defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.220 Memory-Resident

The process of managing the implementation in such a way as to provide an upper bound on memory access times.

3.221 Message

In the context of programmatic message passing, information that can be transferred between processes or threads by being added to and removed from a message queue. A message consists of a fixed-size message buffer.

3.222 Message Catalog

In the context of providing natural language messages to the user, a file or storage area containing program messages, command prompts, and responses to prompts for a particular native language, territory, and codeset.

3.223 Message Catalog Descriptor

In the context of providing natural language messages to the user, a per-process unique value used to identify an open message catalog. A message catalog descriptor may be implemented using a file descriptor.

3.224 Message Queue

In the context of programmatic message passing, an object to which messages can be added and removed. Messages may be removed in the order in which they were added or in priority order.

3.225 Mode

A collection of attributes that specifies a file's type and its access permissions.

Note: File Access Permissions are defined in detail in Section 4.4 (on page 95).

3.226 Monotonic Clock

A clock whose value cannot be set via *clock_settime*( ) and which cannot have negative clock jumps.

3.227 Mount Point

Either the system root directory or a directory for which the *st_dev* field of structure **stat** differs from that of its parent directory.

Note: The **stat** structure is defined in detail in **<sys/stat.h>**.

3.228 Multi-Character Collating Element

A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*.

3.229 Mutex

A synchronization object used to allow multiple threads to serialize their access to shared data. The name derives from the capability it provides; namely, mutual-exclusion. The thread that has locked a mutex becomes its owner and remains the owner until that same thread unlocks the mutex.

3.230 Name

In the shell command language, a word consisting solely of underscores, digits, and alphabetics from the portable character set. The first character of a name is not a digit.

Note: The portable character set is defined in detail in Section 6.1 (on page 111).

3.231 Named STREAM

A STREAMS-based file descriptor that is attached to a name in the file system name space. All subsequent operations on the named STREAM act on the STREAM that was associated with the file descriptor until the name is disassociated from the STREAM.

3.232 NaN (Not a Number)

A set of values that may be stored in a floating type but that are neither Inf nor valid floating-point numbers. Not all systems support NaN values.

3.233 Native Language

A computer user's spoken or written language, such as American English, British English, Danish, Dutch, French, German, Italian, Japanese, Norwegian, or Swedish.

3.234 Negative Response

An input string that matches one of the responses acceptable to the *LC_MESSAGES* category keyword **noexpr**, matching an extended regular expression in the current locale.

Note: The *LC_MESSAGES* category is defined in detail in Section 7.3.6 (on page 148).

3.235 Network

A collection of interconnected hosts.

Note: The term network in IEEE Std 1003.1-200x is used to refer to the network of hosts. The term batch system is used to refer to the network of batch servers.

3.236 Network Address

A network-visible identifier used to designate specific endpoints in a network. Specific endpoints on host systems have addresses, and host systems may also have addresses.

3.237 Network Byte Order

The way of representing any **int** type such that, when transmitted over a network via a network endpoint, the **int** type is transmitted as an appropriate number of octets with the most significant octet first, followed by any other octets in descending order of significance.

Note: This order is more commonly known as big-endian ordering. See also Section 4.8 (on page 97).

3.238 Newline Character (<newline>)

A character that in the output stream indicates that printing should start at the beginning of the next line. It is the character designated by `'\n'` in the C language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line.

3.239 Nice Value

A number used as advice to the system to alter process scheduling. Numerically smaller values give a process additional preference when scheduling a process to run. Numerically larger values reduce the preference and make a process less likely to run. Typically, a process with a smaller nice value runs to completion more quickly than an equivalent process with a higher nice value. The symbol {NZERO} specifies the default nice value of the system.

3.240 Non-Blocking

A property of an open file description that causes function calls involving it to return without delay when it is detected that the requested action associated with the function call cannot be completed without unknown delay.

Note: The exact semantics are dependent on the type of file associated with the open file description. For data reads from devices such as ttys and FIFOs, this property causes the read to return immediately when no data was available. Similarly, for writes, it causes the call to return immediately when the thread would otherwise be delayed in the write operation; for example, because no space was available. For networking, it causes functions not to await protocol events (for example, acknowledgements) to occur. See also the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.10.7, Socket I/O Mode.

3.241 Non-Spacing Characters

A character, such as a character representing a diacritical mark in the ISO/IEC 6937:1994 standard coded character set, which is used in combination with other characters to form composite graphic symbols.

3.242 NUL

A character with all bits set to zero.

3.243 Null Byte

A byte with all bits set to zero.

3.244 Null Pointer

The value that is obtained by converting the number 0 into a pointer; for example, (**void** *) 0. The C language guarantees that this value does not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

3.245 Null String

See *Empty String* in Section 3.145 (on page 52).

3.246 Null Wide-Character Code

A wide-character code with all bits set to zero.

3.247 Number Sign

The character `'#'`, also known as *hash sign*.

3.248 Object File

A regular file containing the output of a compiler, formatted as input to a linkage editor for linking with other object files into an executable form. The methods of linking are unspecified and may involve the dynamic linking of objects at runtime. The internal format of an object file is unspecified, but a conforming application cannot assume an object file is a text file.

3.249 Octet

Unit of data representation that consists of eight contiguous bits.

3.250 Offset Maximum

An attribute of an open file description representing the largest value that can be used as a file offset.

3.251 Opaque Address

An address such that the entity making use of it requires no details about its contents or format.

3.252 Open File

A file that is currently associated with a file descriptor.

3.253 Open File Description

A record of how a process or group of processes is accessing a file. Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor. A file offset, file status, and file access modes are attributes of an open file description.

3.254 Operand

An argument to a command that is generally used as an object supplying information to a utility necessary to complete its processing. Operands generally follow the options in a command line.

Note: Utility Argument Syntax is defined in detail in Section 12.1 (on page 197).

3.255 Operator

In the shell command language, either a control operator or a redirection operator.

3.256 Option

An argument to a command that is generally used to specify changes in the utility's default behavior.

Note: Utility Argument Syntax is defined in detail in Section 12.1 (on page 197).

3.257 Option-Argument

A parameter that follows certain options. In some cases an option-argument is included within the same argument string as the option—in most cases it is the next argument.

Note: Utility Argument Syntax is defined in detail in Section 12.1 (on page 197).

3.258 Orientation

A stream has one of three orientations: unoriented, byte-oriented, or wide-oriented.

Note: For further information, see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.5.2, Stream Orientation and Encoding Rules.

3.259 Orphaned Process Group

A process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

3.260 Page

The granularity of process memory mapping or locking.

Physical memory and memory objects can be mapped into the address space of a process on page boundaries and in integral multiples of pages. Process address space can be locked into memory (made memory-resident) on page boundaries and in integral multiples of pages.

3.261 Page Size

The size, in bytes, of the system unit of memory allocation, protection, and mapping. On systems that have segment rather than page-based memory architectures, the term page means a segment.

3.262 Parameter

In the shell command language, an entity that stores values. There are three types of parameters: variables (named parameters), positional parameters, and special parameters. Parameter expansion is accomplished by introducing a parameter with the `'$'` character.

Note: See also the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.5, Parameters and Variables.

In the C language, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition.

3.263 Parent Directory

When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the pathname dot-dot in the given directory.

When discussing other types of files, a directory containing a directory entry for the file under discussion.

This concept does not apply to dot and dot-dot.

3.264 Parent Process

The process which created (or inherited) the process under discussion.

3.265 Parent Process ID

An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-defined system process.

3.266 Pathname

A character string that is used to identify a file. In the context of IEEE Std 1003.1-200x, a pathname consists of, at most, {PATH_MAX} bytes, including the terminating null byte. It has an optional beginning slash, followed by zero or more filenames separated by slashes. A pathname may optionally contain one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash.

Note: Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.267 Pathname Component

See *Filename* in Section 3.169 (on page 56).

3.268 Path Prefix

A pathname, with an optional ending slash, that refers to a directory.

3.269 Pattern

A sequence of characters used either with regular expression notation or for pathname expansion, as a means of selecting various character strings or pathnames, respectively.

Note: Regular Expressions are defined in detail in Chapter 9 (on page 165).

See also the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.6, Pathname Expansion.

The syntaxes of the two types of patterns are similar, but not identical; IEEE Std 1003.1-200x always indicates the type of pattern being referred to in the immediate context of the use of the term.

3.270 Period

The character `'.'`. The term period is contrasted with dot (see also Section 3.135 (on page 51)), which is used to describe a specific directory entry.

3.271 Permissions

Attributes of an object that determine the privilege necessary to access or manipulate the object.

Note: File Access Permissions are defined in detail in Section 4.4 (on page 95).

3.272 Persistence

A mode for semaphores, shared memory, and message queues requiring that the object and its state (including data, if any) are preserved after the object is no longer referenced by any process.

Persistence of an object does not imply that the state of the object is maintained across a system crash or a system reboot.

3.273 Pipe

An object accessed by one of the pair of file descriptors created by the *pipe*( ) function. Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy.

Note: The *pipe*( ) function is defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.274 Polling

A scheduling scheme whereby the local process periodically checks until the prespecified events (for example, read, write) have occurred.

3.275 Portable Character Set

The collection of characters that are required to be present in all locales supported by conforming systems.

Note: The portable character set is defined in detail in Section 6.1 (on page 111).

This term is contrasted against the smaller *portable filename character set*; see also Section 3.276.

3.276 Portable Filename Character Set

The set of characters from which portable filenames are constructed.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore, and hyphen characters, respectively.

3.277 Positional Parameter

In the shell command language, a parameter denoted by a single digit or one or more digits in curly braces.

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.5.1, Positional Parameters.

3.278 Preallocation

The reservation of resources in a system for a particular use.

Preallocation does not imply that the resources are immediately allocated to that use, but merely indicates that they are guaranteed to be available in bounded time when needed.

3.279 Preempted Process (or Thread)

A running thread whose execution is suspended due to another thread becoming runnable at a higher priority.

3.280 Previous Job

In the context of job control, the job that will be used as the default for the *fg* or *bg* utilities if the current job exits. There is at most one previous job; see also Section 3.203 (on page 60).

3.281 Printable Character

One of the characters included in the **print** character classification of the *LC_CTYPE* category in the current locale.

Note: The *LC_CTYPE* category is defined in detail in Section 7.3.1 (on page 122).

3.282 Printable File

A text file consisting only of the characters included in the **print** and **space** character classifications of the *LC_CTYPE* category and the <backspace>, all in the current locale.

Note: The *LC_CTYPE* category is defined in detail in Section 7.3.1 (on page 122).

3.283 Priority

A non-negative integer associated with processes or threads whose value is constrained to a range defined by the applicable scheduling policy. Numerically higher values represent higher priorities.

3.284 Priority Band

The queuing order applied to normal priority STREAMS messages. High priority STREAMS messages are not grouped by priority bands. The only differentiation made by the STREAMS mechanism is between zero and non-zero bands, but specific protocol modules may differentiate between priority bands.

3.285 Priority Inversion

A condition in which a thread that is not voluntarily suspended (waiting for an event or time delay) is not running while a lower priority thread is running. Such blocking of the higher priority thread is often caused by contention for a shared resource.

3.286 Priority Scheduling

A performance and determinism improvement facility to allow applications to determine the order in which threads that are ready to run are granted access to processor resources.

3.287 Priority-Based Scheduling

Scheduling in which the selection of a running thread is determined by the priorities of the runnable processes or threads.

3.288 Privilege

See *Appropriate Privileges* in Section 3.19 (on page 35).

3.289 Process

An address space with one or more threads executing within that address space, and the required system resources for those threads.

Note: Many of the system resources defined by IEEE Std 1003.1-200x are shared among all of the threads within a process. These include the process ID, the parent process ID, process group ID, session membership, real, effective, and saved-set user ID, real, effective, and saved-set group ID, supplementary group IDs, current working directory, root directory, file mode creation mask, and file descriptors.

3.290 Process Group

A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by a process group ID. A newly created process joins the process group of its creator.

3.291 Process Group ID

The unique positive integer identifier representing a process group during its lifetime.

Note: See also Process Group ID Reuse defined in Section 4.12 (on page 99).

3.292 Process Group Leader

A process whose process ID is the same as its process group ID.

3.293 Process Group Lifetime

A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, due either to the end of the last process' lifetime or to the last remaining process calling the *setsid*( ) or *setpgid*( ) functions.

Note: The *setsid*( ) and *setpgid*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.294 Process ID

The unique positive integer identifier representing a process during its lifetime.

Note: See also Process ID Reuse defined in Section 4.12 (on page 99).

3.295 Process Lifetime

The period of time that begins when a process is created and ends when its process ID is returned to the system. After a process is created with a *fork*( ) function, it is considered active. At least one thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a *wait*( ), *waitid*( ), or *waitpid*( ) function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

Note: The *fork*( ), *wait*( ), *waitid*( ), and *waitpid*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.296 Process Memory Locking

A performance improvement facility to bind application programs into the high-performance random access memory of a computer system. This avoids potential latencies introduced by the operating system in storing parts of a program that were not recently referenced on secondary memory devices.

3.297 Process Termination

There are two kinds of process termination:

1. Normal termination occurs by a return from *main*( ) or when requested with the *exit*( ) or *_exit*( ) functions.
2. Abnormal termination occurs when requested by the *abort*( ) function or when some signals are received.

Note: The *_exit*( ), *abort*( ), and *exit*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.298 Process-To-Process Communication

The transfer of data between processes.

3.299 Process Virtual Time

The measurement of time in units elapsed by the system clock while a process is executing.

3.300 Program

A prepared sequence of instructions to the system to accomplish a defined task. The term program in IEEE Std 1003.1-200x encompasses applications written in the Shell Command Language, complex utility input languages (for example, *awk*, *lex*, *sed*, and so on), and high-level languages.

3.301 Protocol

A set of semantic and syntactic rules for exchanging information.

3.302 Pseudo-Terminal

A facility that provides an interface that is identical to the terminal subsystem. A pseudo-terminal is composed of two devices: the *master device* and a *slave device*. The slave device provides processes with an interface that is identical to the terminal interface, although there need not be hardware behind that interface. Anything written on the master device is presented to the slave as an input and anything written on the slave device is presented as an input on the master side.

3.303 Radix Character

The character that separates the integer part of a number from the fractional part.

3.304 Read-Only File System

A file system that has implementation-defined characteristics restricting modifications.

Note: File Times Update is described in detail in Section 4.7 (on page 96).

3.305 Read-Write Lock

Multiple readers, single writer (read-write) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have write access at any given time. They are typically used to protect data that is read-only more frequently than it is changed.

Read-write locks can be used to synchronize threads in the current process and other processes if they are allocated in memory that is writable and shared among the cooperating processes and have been initialized for this behavior.

3.306 Real Group ID

The attribute of a process that, at the time of process creation, identifies the group of the user who created the process; see also Section 3.188 (on page 58).

3.307 Real Time

Time measured as total units elapsed by the system clock without regard to which thread is executing.

3.308 Realtime Signal Extension

A determinism improvement facility to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal functions.

3.309 Real User ID

The attribute of a process that, at the time of process creation, identifies the user who created the process; see also Section 3.425 (on page 91).

3.310 Record

A collection of related data units or words which is treated as a unit. |

3.311 Redirection |

In the shell command language, a method of associating files with the input or output of commands. |

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.7, Redirection.

3.312 Redirection Operator

In the shell command language, a token that performs a redirection function. It is one of the following symbols:

```
<     >     >|     <<     >>     <&     >&     <<-     <>
```

3.313 Reentrant Function

A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved.

3.314 Referenced Shared Memory Object

A shared memory object that is open or has one or more mappings defined on it.

3.315 Refresh

To ensure that the information on the user's terminal screen is up-to-date.

3.316 Regular Expression

A pattern that selects specific strings from a set of character strings.

Note: Regular Expressions are described in detail in Chapter 9 (on page 165).

3.317 Region

In the context of the address space of a process, a sequence of addresses.

In the context of a file, a sequence of offsets.

3.318 Regular File

A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system.

3.319 Relative Pathname

A pathname not beginning with a slash.

Note: Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.320 Relocatable File

A file holding code or data suitable for linking with other object files to create an executable or a shared object file.

3.321 Relocation

The process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction transfers control to the proper destination address at execution.

3.322 Requested Batch Service

A service that is either rejected or performed prior to a response from the service to the requester.

3.323 (Time) Resolution

The minimum time interval that a clock can measure or whose passage a timer can detect.

3.324 Root Directory

A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash.

3.325 Runnable Process (or Thread)

A thread that is capable of being a running thread, but for which no processor is available.

3.326 Running Process (or Thread)

A thread currently executing on a processor. On multi-processor systems there may be more than one such thread in a system at a time.

3.327 Saved Resource Limits

An attribute of a process that provides some flexibility in the handling of unrepresentable resource limits, as described in the *exec* family of functions and *setrlimit*( ).

Note: The *exec* and *setrlimit*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.328 Saved Set-Group-ID

An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the *exec* family of functions and *setgid*( ).

Note: The *exec* and *setgid*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.329 Saved Set-User-ID

An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in the *exec* family of functions and *setuid*( ).

Note: The *exec* and *setuid*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.330 Scheduling

The application of a policy to select a runnable process or thread to become a running process or thread, or to alter one or more of the thread lists.

3.331 Scheduling Allocation Domain

The set of processors on which an individual thread can be scheduled at any given time.

3.332 Scheduling Contention Scope

A property of a thread that defines the set of threads against which that thread competes for resources.

For example, in a scheduling decision, threads sharing scheduling contention scope compete for processor resources. In IEEE Std 1003.1-200x, a thread has scheduling contention scope of either PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS.

3.333 Scheduling Policy

A set of rules that is used to determine the order of execution of processes or threads to achieve some goal.

Note: Scheduling Policy is defined in detail in Section 4.13 (on page 99).

3.334 Screen

A rectangular region of columns and lines on a terminal display. A screen may be a portion of a physical display device or may occupy the entire physical area of the display device.

3.335 Scroll

To move the representation of data vertically or horizontally relative to the terminal screen. There are two types of scrolling:

1. The cursor moves with the data.
2. The cursor remains stationary while the data moves.

3.336 Semaphore

A minimum synchronization primitive to serve as a basis for more complex synchronization mechanisms to be defined by the application program.

Note: Semaphores are defined in detail in Section 4.15 (on page 100).

3.337 Session

A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see *setsid*( ). There can be multiple process groups in the same session.

Note: The *setsid*( ) function is defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.338 Session Leader

A process that has created a session.

Note: For further information, see the *setsid*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x.

3.339 Session Lifetime

The period between when a session is created and the end of the lifetime of all the process groups that remain as members of the session.

3.340 Shared Memory Object

An object that represents memory that can be mapped concurrently into the address space of more than one process.

3.341 Shell

A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal.

3.342 Shell, the

The Shell Command Language Interpreter; a specific instance of a shell.

Note: For further information, see the *sh* utility defined in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.343 Shell Script

A file containing shell commands. If the file is made executable, it can be executed by specifying its name as a simple command. Execution of a shell script causes a shell to execute the commands within the script. Alternatively, a shell can be requested to execute the commands in a shell script by specifying the name of the shell script as the operand to the *sh* utility.

Note: Simple Commands are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1, Simple Commands.

The *sh* utility is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.344 Signal

A mechanism by which a process or thread may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself.

3.345 Signal Stack

Memory established for a thread, in which signal handlers catching signals sent to that thread are executed.

3.346 Single-Quote

The character `'''`, also known as *apostrophe*.

3.347 Slash

The character `'/'`, also known as *solidus*.

3.348 Socket

A file of a particular type that is used as a communications endpoint for process-to-process communication as described in the System Interfaces volume of IEEE Std 1003.1-200x.

3.349 Socket Address

An address associated with a socket or remote endpoint, including an address family identifier and addressing information specific to that address family. The address may include multiple parts, such as a network address associated with a host system and an identifier for a specific endpoint.

3.350 Soft Limit

A resource limitation established for each process that the process may set to any value less than or equal to the hard limit. |

3.351 Source Code |

When dealing with the Shell Command Language, input to the command language interpreter. | The term shell script is synonymous with this meaning.

When dealing with an ISO/IEC-conforming programming language, source code is input to a compiler conforming to that ISO/IEC standard.

Source code also refers to the input statements prepared for the following standard utilities: *awk*, *bc*, *ed*, *lex*, *localedef*, *make*, *sed*, and *yacc*.

Source code can also refer to a collection of sources meeting any or all of these meanings.

Note: The *awk*, *bc*, *ed*, *lex*, *localedef*, *make*, *sed*, and *yacc* utilities are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.352 Space Character (<space>)

The character defined in the portable character set as <space>. The <space> is a member of the **space** character class of the current locale, but represents the single character, and not all of the possible members of the class; see also Section 3.431 (on page 92).

3.353 Spawn

A process creation primitive useful for systems that have difficulty with *fork*( ) and as an efficient replacement for *fork*( )/*exec*.

3.354 Special Built-In

See *Built-In Utility* in Section 3.83 (on page 44).

3.355 Special Parameter

In the shell command language, a parameter named by a single character from the following list:

```
*    @    #    ?    !    -    $    0
```

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.5.2, Special Parameters.

3.356 Spin Lock

A synchronization object used to allow multiple threads to serialize their access to shared data.

3.357 Sporadic Server

A scheduling policy for threads and processes that reserves a certain amount of execution capacity for processing aperiodic events at a given priority level.

3.358 Standard Error

An output stream usually intended to be used for diagnostic messages.

3.359 Standard Input

An input stream usually intended to be used for primary data input.

3.360 Standard Output

An output stream usually intended to be used for primary data output.

3.361 Standard Utilities

The utilities described in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.362 Stream

Appearing in lowercase, a stream is a file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the *fdopen*( ), *fopen*( ), or *popen*( ) functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output; see also Section 3.363.

Note: For further information, see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.5, Standard I/O Streams.

The *fdopen*( ), *fopen*( ), or *popen*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.363 STREAM

Appearing in uppercase, STREAM refers to a full duplex connection between a process and an open device or pseudo-device. It optionally includes one or more intermediate processing modules that are interposed between the process end of the STREAM and the device driver (or pseudo-device driver) end of the STREAM; see also Section 3.362.

Note: For further information, see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.6, STREAMS.

3.364 STREAM End

The STREAM end is the driver end of the STREAM and is also known as the downstream end of the STREAM.

3.365 STREAM Head

The STREAM head is the beginning of the STREAM and is at the boundary between the system and the application process. This is also known as the upstream end of the STREAM.

3.366 STREAMS Multiplexor

A driver with multiple STREAMS connected to it. Multiplexing with STREAMS connected above is referred to as N-to-1, or *upper multiplexing*. Multiplexing with STREAMS connected below is referred to as 1-to-N or *lower multiplexing*.

3.367 String

A contiguous sequence of bytes terminated by and including the first null byte.

3.368 Subshell

A shell execution environment, distinguished from the main or current shell execution environment.

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.12, Shell Execution Environment.

3.369 Successfully Transferred

For a write operation to a regular file, when the system ensures that all data written is readable on any subsequent open of the file (even one that follows a system or power failure) in the absence of a failure of the physical storage medium.

For a read operation, when an image of the data on the physical storage medium is available to the requesting process.

3.370 Supplementary Group ID

An attribute of a process used in determining file access permissions. A process has up to {NGROUPS_MAX} supplementary group IDs in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created.

3.371 Suspended Job

A job that has received a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal that caused the process group to stop. A suspended job is a background job, but a background job is not necessarily a suspended job.

3.372 Symbolic Link

A type of file with the property that when the file is encountered during pathname resolution, a string stored by the file is used to modify the pathname resolution. The stored string has a length of {SYMLINK_MAX} bytes or fewer.

Note: Pathname Resolution is defined in detail in Section 4.11 (on page 98).

3.373 Synchronized Input and Output

A determinism and robustness improvement mechanism to enhance the data input and output mechanisms, so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices.

3.374 Synchronized I/O Completion

The state of an I/O operation that has either been successfully transferred or diagnosed as unsuccessful.

3.375 Synchronized I/O Data Integrity Completion

For read, when the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests are successfully transferred prior to reading the data.

For write, when the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (access time, modification time, status change time) need not be successfully transferred prior to returning to the calling process.

3.376 Synchronized I/O File Integrity Completion

Identical to a synchronized I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including access time, modification time, status change time) are successfully transferred prior to returning to the calling process.

3.377 Synchronized I/O Operation

An I/O operation performed on a file that provides the application assurance of the integrity of its data and files.

3.378 Synchronous I/O Operation

An I/O operation that causes the thread requesting the I/O to be blocked from further use of the processor until that I/O operation completes.

Note: A synchronous I/O operation does not imply synchronized I/O data integrity completion or synchronized I/O file integrity completion.

3.379 Synchronously-Generated Signal

A signal that is attributable to a specific thread.

For example, a thread executing an illegal instruction or touching invalid memory causes a synchronously-generated signal. Being synchronous is a property of how the signal was generated and not a property of the signal number.

3.380 System

An implementation of IEEE Std 1003.1-200x.

3.381 System Crash

An interval initiated by an unspecified circumstance that causes all processes (possibly other than special system processes) to be terminated in an undefined manner, after which any changes to the state and contents of files created or written to by an application prior to the interval are undefined, except as required elsewhere in IEEE Std 1003.1-200x.

3.382 System Console

An implementation-defined device that receives messages sent by the *syslog*( ) function, and the *fmtmsg*( ) function when the MM_CONSOLE flat is set.

Note: The *syslog*( ) and *fmtmsg*( ) functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.383 System Databases

An implementation provides two system databases.

The *group database* contains the following information for each group:

1. Group name
2. Numerical group ID
3. List of all users allowed in the group

The *user database* contains the following information for each user:

1. User name
2. Numerical user ID
3. Numerical group ID
4. Initial working directory
5. Initial user program

If the initial user program field is null, the system default is used. If the initial working directory field is null, the interpretation of that field is implementation-defined. These databases may contain other fields that are unspecified by IEEE Std 1003.1-200x.

3.384 System Documentation

All documentation provided with an implementation except for the conformance document. Electronically distributed documents for an implementation are considered part of the system documentation.

3.385 System Process

An implementation-defined object, other than a process executing an application, that has a process ID.

3.386 System Reboot

An implementation-defined sequence of events that may result in the loss of transitory data; that is, data that is not saved in permanent storage. For example, message queues, shared memory, semaphores, and processes.

3.387 System Trace Event

A trace event that is generated by the implementation, in response either to a system-initiated action or to an application-requested action, except for a call to *posix_trace_event*( ). When supported by the implementation, a system-initiated action generates a process-independent system trace event and an application-requested action generates a process-dependent system trace event. For a system trace event not defined by IEEE Std 1003.1-200x, the associated trace event type identifier is derived from the implementation-defined name for this trace event, and the associated data is of implementation-defined content and length.

3.388 System-Wide

Pertaining to events occurring in all processes existing in an implementation at a given point in time.

3.389 Tab Character (<tab>)

A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. It is the character designated by `'\t'` in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation.

3.390 Terminal (or Terminal Device)

A character special file that obeys the specifications of the general terminal interface.

Note: The General Terminal Interface is defined in detail in Chapter 11 (on page 183).

3.391 Text Column

A roughly rectangular block of characters capable of being laid out side-by-side next to other text columns on an output page or terminal screen. The widths of text columns are measured in column positions.

3.392 Text File

A file that contains characters organized into one or more lines. The lines do not contain NUL characters and none can exceed {LINE_MAX} bytes in length, including the <newline>. Although IEEE Std 1003.1-200x does not distinguish between text files and binary files (see the ISO C standard), many utilities only produce predictable or meaningful output when operating on text files. The standard utilities that have such restrictions always specify *text files* in their

STDIN or INPUT FILES sections.

3.393 Thread

A single flow of control within a process. Each thread has its own thread ID, scheduling priority and policy, *errno* value, thread-specific key/value bindings, and the required system resources to support a flow of control. Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via *malloc*(), directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.

Note: The *malloc*() function is defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.394 Thread ID

Each thread in a process is uniquely identified during its lifetime by a value of type **pthread_t** called a thread ID.

3.395 Thread List

An ordered set of runnable threads that all have the same ordinal value for their priority.

The ordering of threads on the list is determined by a scheduling policy or policies. The set of thread lists includes all runnable threads in the system.

3.396 Thread-Safe

A function that may be safely invoked concurrently by multiple threads. Each function defined in the System Interfaces volume of IEEE Std 1003.1-200x is thread-safe unless explicitly stated otherwise. Examples are any ''pure'' function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads.

3.397 Thread-Specific Data Key

A process global handle of type **pthread_key_t** which is used for naming thread-specific data.

Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*() and accessed by *pthread_getspecific*() are maintained on a per-thread basis and persist for the life of the calling thread.

Note: The *pthread_getspecific*() and *pthread_setspecific*() functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.398 Tilde

The character `'~'`.

3.399 Timeouts

A method of limiting the length of time an interface will block; see also Section 3.76 (on page 43).

3.400 Timer

A mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.

3.401 Timer Overrun

A condition that occurs each time a timer, for which there is already an expiration signal queued to the process, expires.

3.402 Token

In the shell command language, a sequence of characters that the shell considers as a single unit when reading input. A token is either an operator or a word.

Note: The rules for reading input are defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.3, Token Recognition.

3.403 Trace Analyzer Process

A process that extracts trace events from a trace stream to retrieve information about the behavior of an application.

3.404 Trace Controller Process

A process that creates a trace stream for tracing a process.

3.405 Trace Event

A data object that represents an action executed by the system, and that is recorded in a trace stream.

3.406 Trace Event Type

A data object type that defines a class of trace event.

3.407 Trace Event Type Mapping

A one-to-one mapping between trace event types and trace event names.

3.408 Trace Filter

A filter that allows the trace controller process to specify those trace event types that are to be ignored; that is, not generated.

3.409 Trace Generation Version

A data object that is an implementation-defined character string, generated by the trace system and describing the origin and version of the trace system.

3.410 Trace Log

The flushed image of a trace stream, if the trace stream is created with a trace log.

3.411 Trace Point

An action that may cause a trace event to be generated.

3.412 Trace Stream

An opaque object that contains trace events plus internal data needed to interpret those trace events.

3.413 Trace Stream Identifier

A handle to manage tracing operations in a trace stream.

3.414 Trace System

A system that allows both system and user trace events to be generated into a trace stream. These trace events can be retrieved later.

3.415 Traced Process

A process for which at least one trace stream has been created. A traced process is also called a target process.

3.416 Tracing Status of a Trace Stream

A status that describes the state of an active trace stream. The tracing status of a trace stream can be retrieved from the trace stream attributes. An active trace stream can be in one of two states: running or suspended.

3.417 Typed Memory Name Space

A system-wide name space that contains the names of the typed memory objects present in the system. It is configurable for a given implementation.

3.418 Typed Memory Object

A combination of a typed memory pool and a typed memory port. The entire contents of the pool are accessible from the port. The typed memory object is identified through a name that belongs to the typed memory name space.

3.419 Typed Memory Pool

An extent of memory with the same operational characteristics. Typed memory pools may be contained within each other.

3.420 Typed Memory Port

A hardware access path to one or more typed memory pools.

3.421 Unbind

Remove the association between a network address and an endpoint.

3.422 Unit Data

See *Datagram* in Section 3.123 (on page 49).

3.423 Upshifting

The conversion of a lowercase character that has a single-character uppercase representation into this uppercase representation.

3.424 User Database

A system database of implementation-defined format that contains at least the following information for each user ID:

- User name
- Numerical user ID
- Initial numerical group ID
- Initial working directory
- Initial user program

The initial numerical group ID is used by the *newgrp* utility. Any other circumstances under which the initial values are operative are implementation-defined.

If the initial user program field is null, an implementation-defined program is used.

If the initial working directory field is null, the interpretation of that field is implementation-defined.

Note: The *newgrp* utility is defined in detail in the Shell and Utilities volume of IEEE Std 1003.1-200x.

3.425 User ID

A non-negative integer that is used to identify a system user. When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or a saved set-user-ID.

3.426 User Name

A string that is used to identify a user; see also Section 3.424 (on page 90). To be portable across systems conforming to IEEE Std 1003.1-200x, the value is composed of characters from the portable filename character set. The hyphen should not be used as the first character of a portable user name.

3.427 User Trace Event

A trace event that is generated explicitly by the application as a result of a call to *posix_trace_event*( ).

3.428 Utility

A program, excluding special built-in utilities provided as part of the Shell Command Language, that can be called by name from a shell to perform a specific task, or related set of tasks.

Note: For further information on special built-in utilities, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.14, Special Built-In Utilities.

3.429 Variable

In the shell command language, a named parameter.

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.5, Parameters and Variables.

3.430 Vertical-Tab Character (<vertical-tab>)

A character that in the output stream indicates that printing should start at the next vertical tabulation position. It is the character designated by `'\v'` in the C language. If the current position is at or past the last defined vertical tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation.

3.431 White Space

A sequence of one or more characters that belong to the **space** character class as defined via the *LC_CTYPE* category in the current locale.

In the POSIX locale, white space consists of one or more <blank>s (<space>s and <tab>s), <newline>s, <carriage-return>s, <form-feed>s, and <vertical-tab>s.

3.432 Wide-Character Code (C Language)

An integer value corresponding to a single graphic symbol or control code.

Note: C Language Wide-Character Codes are defined in detail in Section 6.3 (on page 115).

3.433 Wide-Character Input/Output Functions

The functions that perform wide-oriented input from streams or wide-oriented output to streams: *fgetwc*( ), *fputwc*( ), *fputws*( ), *fwprintf*( ), *fwscanf*( ), *getwc*( ), *getwchar*( ), *getws*( ), *putwc*( ), *putwchar*( ), *ungetwc*( ), *vfwprintf*( ), *vwprintf*( ), *wprintf*( ), and *wscanf*( ).

Note: These functions are defined in detail in the System Interfaces volume of IEEE Std 1003.1-200x.

3.434 Wide-Character String

A contiguous sequence of wide-character codes terminated by and including the first null wide-character code.

3.435 Word

In the shell command language, a token other than an operator. In some cases a word is also a portion of a word token: in the various forms of parameter expansion, such as ${*name*–*word*}, and variable assignment, such as *name*=*word*, the word is the portion of the token depicted by *word*. The concept of a word is no longer applicable following word expansions—only fields remain.

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.2, Parameter Expansion and the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions.

3.436 Working Directory (or Current Working Directory)

A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash.

3.437 Worldwide Portability Interface

Functions for handling characters in a codeset-independent manner.

3.438 Write

To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term write; see the distinction between display and write in Section 3.132 (on page 50).

3.439 XSI

The X/Open System Interface is the core application programming interface for C and *sh* programming for systems conforming to the Single UNIX Specification. This is a superset of the mandatory requirements for conformance to IEEE Std 1003.1-200x.

3.440 XSI-Conformant

A system which allows an application to be built using a set of services that are consistent across all systems that conform to IEEE Std 1003.1-200x and that support the XSI extension.

Note: See also Chapter 2 (on page 15).

3.441 Zombie Process

A process that has terminated and that is deleted when its exit status has been reported to another process which is waiting for that process to terminate.

3.442 ±0

The algebraic sign provides additional information about any variable that has the value zero when the representation allows the sign to be determined.

Chapter 4

General Concepts

For the purposes of IEEE Std 1003.1-200x, the general concepts given in Chapter 4 apply.

Note: No shading to denote extensions or options occurs in this chapter. Where the terms and definitions given in this chapter are used elsewhere in text related to extensions and options, they are shaded as appropriate.

4.1 Concurrent Execution

Functions that suspend the execution of the calling thread shall not cause the execution of other threads to be indefinitely suspended.

4.2 Directory Protection

If a directory is writable and the mode bit S_ISVTX is set on the directory, a process may remove or rename files within that directory only if one or more of the following is true:

- The effective user ID of the process is the same as that of the owner ID of the file.
- The effective user ID of the process is the same as that of the owner ID of the directory.
- The process has appropriate privileges.

If the S_ISVTX bit is set on a non-directory file, the behavior is unspecified.

4.3 Extended Security Controls

An implementation may provide implementation-defined extended security controls (see Section 3.159 (on page 54)). These permit an implementation to provide security mechanisms to implement different security policies than those described in IEEE Std 1003.1-200x. These mechanisms shall not alter or override the defined semantics of any of the interfaces in IEEE Std 1003.1-200x.

4.4 File Access Permissions

The standard file access control mechanism uses the file permission bits, as described below.

Implementations may provide *additional* or *alternate* file access control mechanisms, or both. An additional access control mechanism shall only further restrict the access permissions defined by the file permission bits. An alternate file access control mechanism shall:

- Specify file permission bits for the file owner class, file group class, and file other class of that file, corresponding to the access permissions.
- Be enabled only by explicit user action, on a per-file basis by the file owner or a user with the appropriate privilege.
- Be disabled for a file after the file permission bits are changed for that file with *chmod*( ). The disabling of the alternate mechanism need not disable any additional mechanisms supported

by an implementation.

Whenever a process requests file access permission for read, write, or execute/search, if no additional mechanism denies access, access shall be determined as follows:

- If a process has the appropriate privilege:
  - — If read, write, or directory search permission is requested, access shall be granted.
  - — If execute permission is requested, access shall be granted if execute permission is granted to at least one user by the file permission bits or by an alternate access control mechanism; otherwise, access shall be denied.
- Otherwise:
  - — The file permission bits of a file contain read, write, and execute/search permissions for the file owner class, file group class, and file other class.
  - — Access shall be granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access shall be denied.

4.5 File Hierarchy

Files in the system are organized in a hierarchical structure in which all of the non-terminal nodes are directories and all of the terminal nodes are any other type of file. Since multiple directory entries may refer to the same file, the hierarchy is properly described as a *directed graph*.

4.6 Filenames

For a filename to be portable across implementations conforming to IEEE Std 1003.1-200x, it shall consist only of the portable filename character set as defined in Section 3.276 (on page 70).

The hyphen character shall not be used as the first character of a portable filename. Uppercase and lowercase letters shall retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used.

4.7 File Times Update

Each file has three distinct associated time values: *st_atime*, *st_mtime*, and *st_ctime*. The *st_atime* field is associated with the times that the file data is accessed; *st_mtime* is associated with the times that the file data is modified; and *st_ctime* is associated with the times that the file status is changed. These values are returned in the file characteristics structure, as described in **<sys/stat.h>**.

Each function or utility in IEEE Std 1003.1-200x that reads or writes data or changes file status indicates which of the appropriate time-related fields shall be ‘‘marked for update’’. If an implementation of such a function or utility marks for update a time-related field not specified by IEEE Std 1003.1-200x, this shall be documented, except that any changes caused by pathname resolution need not be documented. For the other functions or utilities in IEEE Std 1003.1-200x (those that are not explicitly required to read or write file data or change file status, but that in some implementations happen to do so), the effect is unspecified.

An implementation may update fields that are marked for update immediately, or it may update such fields periodically. At an update point in time, any marked fields shall be set to the current time and the update marks shall be cleared. All fields that are marked for update shall be updated when the file ceases to be open by any process, or when a *stat*( ), *fstat*( ), or *lstat*( ) is performed on the file. Other times at which updates are done are unspecified. Marks for update, and updates themselves, are not done for files on read-only file systems; see Section 3.304 (on page 74).

4.8 Host and Network Byte Orders

When data is transmitted over the network, it is sent as a sequence of octets (8-bit unsigned values). If an entity (such as an address or a port number) can be larger than 8 bits, it needs to be stored in several octets. The convention is that all such values are stored with 8 bits in each octet, and with the first (lowest-addressed) octet holding the most-significant bits. This is called ''network byte order''.

Network byte order may not be convenient for processing actual values. For this, it is more sensible for values to be stored as ordinary integers. This is known as ''host byte order''. In host byte order:

- The most significant bit might not be stored in the first byte in address order.
- Bits might not be allocated to bytes in any obvious order at all.

8-bit values stored in **uint8_t** objects do not require conversion to or from host byte order, as they have the same representation. 16 and 32-bit values can be converted using the *htonl*( ), *htons*( ), *ntohl*( ), and *ntohs*( ) functions. When reading data that is to be converted to host byte order, it should either be received directly into a **uint16_t** or **uint32_t** object or should be copied from an array of bytes using *memcpy*( ) or similar. Passing the data through other types could cause the byte order to be changed. Similar considerations apply when sending data.

4.9 Measurement of Execution Time

The mechanism used to measure execution time shall be implementation-defined. The implementation shall also define to whom the CPU time that is consumed by interrupt handlers and system services on behalf of the operating system will be charged. See Section 3.117 (on page 49).

4.10 Memory Synchronization

Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads. The following functions synchronize memory with respect to other threads:

fork( )
pthread_barrier_wait( )
pthread_cond_broadcast( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_create( )
pthread_join( )
pthread_mutex_lock( )
pthread_mutex_timedlock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
pthread_spin_lock( )
pthread_spin_trylock( )
pthread_spin_unlock( )
pthread_rwlock_rdlock( )
pthread_rwlock_timedrdlock( )
pthread_rwlock_timedwrlock( )
pthread_rwlock_tryrdlock( )
pthread_rwlock_trywrlock( )
pthread_rwlock_unlock( )
pthread_rwlock_wrlock( )
sem_post( )
sem_trywait( )
sem_wait( )
wait( )
waitpid( )

Unless explicitly stated otherwise, if one of the above functions returns an error, it is unspecified whether the invocation causes memory to be synchronized.

Applications may allow more than one thread of control to read a memory location simultaneously.

4.11 Pathname Resolution

Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file.

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment **a/b**, file **b** is located in directory **a**). Pathname resolution shall fail if this cannot be accomplished. If the pathname begins with a slash, the predecessor of the first filename in the pathname shall be taken to be the root directory of the process (such pathnames are referred to as *absolute pathnames*). If the pathname does not begin with a slash, the predecessor of the first filename of the pathname shall be taken to be the current working directory of the process (such pathnames are referred to as *relative pathnames*).

The interpretation of a pathname component is dependent on the value of {NAME_MAX} and _POSIX_NO_TRUNC associated with the path prefix of that component. If any pathname component is longer than {NAME_MAX}, the implementation shall consider this an error.

A pathname that contains at least one non-slash character and that ends with one or more trailing slashes shall be resolved as if a single dot character (`'.'`) were appended to the pathname.

If a symbolic link is encountered during pathname resolution, the behavior shall depend on whether the pathname component is at the end of the pathname and on the function being performed. If all of the following are true, then pathname resolution is complete:

1. This is the last pathname component of the pathname.
2. The pathname has no trailing slash.
3. The function is required to act on the symbolic link itself, or certain arguments direct that the function act on the symbolic link itself.

In all other cases, the system shall prefix the remaining pathname, if any, with the contents of the symbolic link. If the combined length exceeds {PATH_MAX}, and the implementation considers this to be an error, *errno* shall be set to [ENAMETOOLONG] and an error indication shall be returned. Otherwise, the resolved pathname shall be the resolution of the pathname just created. If the resulting pathname does not begin with a slash, the predecessor of the first filename of the pathname is taken to be the directory containing the symbolic link.

If the system detects a loop in the pathname resolution process, it shall set *errno* to [ELOOP] and return an error indication. The same may happen if during the resolution process more symbolic links were followed than the implementation allows. This implementation-defined limit shall not be smaller than {SYMLOOP_MAX}.

The special filename dot shall refer to the directory specified by its predecessor. The special filename dot-dot shall refer to the parent directory of its predecessor directory. As a special case, in the root directory, dot-dot may refer to the root directory itself.

A pathname consisting of a single slash shall resolve to the root directory of the process. A null pathname shall not be successfully resolved. A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash.

4.12 Process ID Reuse

A process group ID shall not be reused by the system until the process group lifetime ends.

A process ID shall not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1.

4.13 Scheduling Policy

A scheduling policy affects process or thread ordering:

- When a process or thread is a running thread and it becomes a blocked thread
- When a process or thread is a running thread and it becomes a preempted thread
- When a process or thread is a blocked thread and it becomes a runnable thread
- When a running thread calls a function that can change the priority or scheduling policy of a process or thread
- In other scheduling policy-defined circumstances

Conforming implementations shall define the manner in which each of the scheduling policies may modify the priorities or otherwise affect the ordering of processes or threads at each of the occurrences listed above. Additionally, conforming implementations shall define in what other circumstances and in what manner each scheduling policy may modify the priorities or affect the ordering of processes or threads.

4.14 Seconds Since the Epoch

A value that approximates the number of seconds that have elapsed since the Epoch. A Coordinated Universal Time name (specified in terms of seconds (*tm_sec*), minutes (*tm_min*), hours (*tm_hour*), days since January 1 of the year (*tm_yday*), and calendar year minus 1900 (*tm_year*)) is related to a time represented as seconds since the Epoch, according to the expression below.

If the year is <1970 or the value is negative, the relationship is undefined. If the year is ≥1970 and the value is non-negative, the value is related to a Coordinated Universal Time name according to the C-language expression, where *tm_sec*, *tm_min*, *tm_hour*, *tm_yday*, and *tm_year* are all integer types:

```
tm_sec + tm_min*60 + tm_hour*3600 + tm_yday*86400 +
    (tm_year-70)*31536000 + ((tm_year-69)/4)*86400 -
    ((tm_year-1)/100)*86400 + ((tm_year+299)/400)*86400
```

The relationship between the actual time of day and the current value for seconds since the Epoch is unspecified.

How any changes to the value of seconds since the Epoch are made to align to a desired relationship with the current actual time are made is implementation-defined. As represented in seconds since the Epoch, each and every day shall be accounted for by exactly 86 400 seconds.

Note: The last three terms of the expression add in a day for each year that follows a leap year starting with the first leap year since the Epoch. The first term adds a day every 4 years starting in 1973, the second subtracts a day back out every 100 years starting in 2001, and the third adds a day back in every 400 years starting in 2001. The divisions in the formula are integer divisions; that is, the remainder is discarded leaving only the integer quotient.

4.15 Semaphore

A minimum synchronization primitive to serve as a basis for more complex synchronization mechanisms to be defined by the application program.

For the semaphores associated with the Semaphores option, a semaphore is represented as a shareable resource that has a non-negative integer value. When the value is zero, there is a (possibly empty) set of threads awaiting the availability of the semaphore.

For the semaphores associated with the X/Open System Interface Extension (XSI), a semaphore is a positive integer (0 through 32767). The *semget*( ) function can be called to create a set or array of semaphores. A semaphore set can contain one or more semaphores up to an implementation-defined value.

Semaphore Lock Operation

An operation that is applied to a semaphore. If, prior to the operation, the value of the semaphore is zero, the semaphore lock operation shall cause the calling thread to be blocked and added to the set of threads awaiting the semaphore; otherwise, the value shall be decremented.

Semaphore Unlock Operation

An operation that is applied to a semaphore. If, prior to the operation, there are any threads in the set of threads awaiting the semaphore, then some thread from that set shall be removed from the set and becomes unblocked; otherwise, the semaphore value shall be incremented.

4.16 Thread-Safety

Refer to the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.9, Threads.

4.17 Tracing

The trace system allows a traced process to have a selection of events created for it. Traces consist of streams of trace event types.

A trace event type is identified on the one hand by a trace event type name, also referenced as a trace event name, and on the other hand by a trace event type identifier. A trace event name is a human-readable string. A trace event type identifier is an opaque identifier used by the trace system. There shall be a one-to-one relationship between trace event type identifiers and trace event names for a given trace stream and also for a given traced process. The trace event type identifier shall be generated automatically from a trace event name by the trace system either when a trace controller process invokes *posix_trace_trid_eventid_open*( ) or when an instrumented application process invokes *posix_trace_eventid_open*( ). Trace event type identifiers are used to filter trace event types, to allow interpretation of user data, and to identify the kind of trace point that generated a trace event.

Each trace event shall be of a particular trace event type, and associated with a trace event type identifier. The execution of a trace point shall generate a trace event if a trace stream has been created and started for the process that executed the trace point and if the corresponding trace event type identifier is not ignored by filtering.

A generated trace event shall be recorded in a trace stream, and optionally also in a trace log if a trace log is associated with the trace stream, except that:

- For a trace stream, if no resources are available for the event, the event is lost.
- For a trace log, if no resources are available for the event, or a flush operation does not succeed, the event is lost.

A trace event recorded in an active trace stream may be retrieved by an application having the appropriate privileges.

A trace event recorded in a trace log may be retrieved by an application having the appropriate privileges after opening the trace log as a pre-recorded trace stream, with the function *posix_trace_open*( ).

When a trace event is reported it is possible to retrieve the following:

- A trace event type identifier
- A timestamp
- The process ID of the traced process, if the trace event is process-dependent
- Any optional trace event data including its length

- If the Threads option is supported, the thread ID, if the trace event is process-dependent
- The program address at which the trace point was invoked

Trace events may be mapped from trace event types to trace event names. One such mapping shall be associated with each trace stream. An active trace stream is associated with a traced process, and also with its children if the Trace Inherit option is supported and also the inheritance policy is set to _POSIX_TRACE_INHERIT. Therefore each traced process has a mapping of the trace event names to trace event type identifiers that have been defined for that process.

Traces can be recorded into either trace streams or trace logs.

The implementation and format of a trace stream are unspecified. A trace stream need not be and generally is not persistent. A trace stream may be either active or pre-recorded:

- An active trace stream is a trace stream that has been created and has not yet been shut down. It can be of one of the two following classes:
  1. An active trace stream without a trace log that was created with the *posix_trace_create*( ) function
  2. If the Trace Log option is supported, an active trace stream with a trace log that was created with the *posix_trace_create_withlog*( ) function
- A pre-recorded trace stream is a trace stream that was opened from a trace log object using the *posix_trace_open*( ) function.

An active trace stream can loop. This behavior means that when the resources allocated by the trace system for the trace stream are exhausted, the trace system reuses the resources associated with the oldest recorded trace events to record new trace events.

If the Trace Log option is supported, an active trace stream with a trace log can be flushed. This operation causes the trace system to write trace events from the trace stream to the associated trace log, following the defined policies or using an explicit function call. After this operation, the trace system may reuse the resources associated with the flushed trace events.

An active trace stream with or without a trace log can be cleared. This operation shall cause all the resources associated with this trace stream to be reinitialized. The trace stream shall behave as if it was returning from its creation, except that the mapping of trace event type identifiers to trace event names shall not be cleared. If a trace log was associated with this trace stream, the trace log shall also be reinitialized.

A trace log shall be recorded when the *posix_trace_shutdown*( ) operation is invoked or during tracing, depending on the tracing strategy which is defined by a log policy. After the trace stream has been shut down, the trace information can be retrieved from the associated trace log using the same interface used to retrieve information from an active trace stream.

For a traced process, if the Trace Inherit option is supported and the trace stream's inheritance attribute is _POSIX_TRACE_INHERIT, the initial targeted traced process shall be traced together with all of its future children. The *posix_pid* member of each trace event in a trace stream shall be the process ID of the traced process.

Each trace point may be an implementation-defined action such as a context switch, or an application-programmed action such as a call to a specific operating system service (for example, *fork*( )) or a call to *posix_trace_event*( ).

Trace points may be filtered. The operation of the filter is to filter out (ignore) selected trace events. By default, no trace events are filtered.

The results of the tracing operations can be analyzed and monitored by a trace controller process or a trace analyzer process.

Only the trace controller process has control of the trace stream it has created. The control of the operation of a trace stream is done using its corresponding trace stream identifier. The trace controller process is able to:

- Initialize the attributes of a trace stream
- Create the trace stream
- Start and stop tracing
- Know the mapping of the traced process
- If the Trace Event Filter option is supported, filter the type of trace events to be recorded
- Shut the trace stream down

A traced process may also be a trace controller process. Only the trace controller process can control its trace stream(s). A trace stream created by a trace controller process shall be shut down if its controller process terminates or executes another file.

A trace controller process may also be a trace analyzer process. Trace analysis can be done concurrently with the traced process or can be done off-line, in the same or in a different platform.

4.18 Treatment of Error Conditions for Mathematical Functions

For all the functions in the **<math.h>** header, an application wishing to check for error situations should set *errno* to 0 and call *feclearexcept*(FE_ALL_EXCEPT) before calling the function. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

The following error conditions are defined for all functions in the **<math.h>** header.

4.18.1 Domain Error

A *domain error* shall occur if an input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.

On a domain error, the function shall return an implementation-defined value; if the integer expression (math_errhandling & MATH_ERRNO) is non-zero, *errno* shall be set to [EDOM]; if the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the ‘‘invalid’’ floating-point exception shall be raised.

4.18.2 Pole Error

A *pole error* occurs if the mathematical result of the function is an exact infinity (for example, log(0.0)).

On a pole error, the function shall return the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL according to the return type, with the same sign as the correct value of the function; if the integer expression (math_errhandling & MATH_ERRNO) is non-zero, *errno* shall be set to [ERANGE]; if the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the ‘‘divide-by-zero’’ floating-point exception shall be raised.

4.18.3 Range Error

A *range error* shall occur if the finite mathematical result of the function cannot be represented in an object of the specified type, due to extreme magnitude.

4.18.3.1 Result Overflows

A floating result overflows if the magnitude of the mathematical result is finite but so large that the mathematical result cannot be represented without extraordinary roundoff error in an object of the specified type. If a floating result overflows and default rounding is in effect, then the function shall return the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL according to the return type, with the same sign as the correct value of the function; if the integer expression (math_errhandling & MATH_ERRNO) is non-zero, *errno* shall be set to [ERANGE]; if the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, the ‘‘overflow’’ floating-point exception shall be raised.

4.18.3.2 Result Underflows

The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type. If the result underflows, the function shall return an implementation-defined value whose magnitude is no greater than the smallest normalized positive number in the specified type; if the integer expression (math_errhandling & MATH_ERRNO) is non-zero, whether *errno* is set to [ERANGE] is implementation-defined; if the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, whether the ‘‘underflow’’ floating-point exception is raised is implementation-defined.

4.19 Treatment of NaN Arguments for the Mathematical Functions

For functions called with a NaN argument, no errors shall occur and a NaN shall be returned, except where stated otherwise.

If a function with one or more NaN arguments returns a NaN result, the result should be the same as one of the NaN arguments (after possible type conversion), except perhaps for the sign.

On implementations that support the IEC 60559:1989 standard floating point, functions with signaling NaN argument(s) shall be treated as if the function were called with an argument that is a required domain error and shall return a quiet NaN result, except where stated otherwise.

Note: The function might never see the signaling NaN, since it might trigger when the arguments are evaluated during the function call.

On implementations that support the IEC 60559:1989 standard floating point, for those functions that do not have a documented domain error, the following shall apply:

These functions shall fail if:

Domain Error Any argument is a signaling NaN.

Either, the integer expression (math_errhandling & MATH_ERRNO) is non-zero and *errno* shall be set to [EDOM], or the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero and the invalid floating-point exception shall be raised.

4.20 Utility

A utility program shall be either an executable file, such as might be produced by a compiler or linker system from computer source code, or a file of shell source code, directly interpreted by the shell. The program may have been produced by the user, provided by the system implementor, or acquired from an independent distributor.

The system may implement certain utilities as shell functions (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.5, Function Definition Command) or built-in utilities, but only an application that is aware of the command search order described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1.1, Command Search and Execution or of performance characteristics can discern differences between the behavior of such a function or built-in utility and that of an executable file.

4.21 Variable Assignment

In the shell command language, a word consisting of the following parts:

```
varname=value
```

When used in a context where assignment is defined to occur and at no other time, the *value* (representing a word or field) shall be assigned as the value of the variable denoted by *varname*.

Note: For further information, see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1, Simple Commands.

The *varname* and *value* parts shall meet the requirements for a name and a word, respectively, except that they are delimited by the embedded unquoted equals-sign, in addition to other delimiters.

Note: Additional delimiters are described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.3, Token Recognition.

When a variable assignment is done, the variable shall be created if it did not already exist. If *value* is not specified, the variable shall be given a null value.

Note: An alternative form of variable assignment:

```
symbol=value
```

(where *symbol* is a valid word delimited by an equals-sign, but not a valid name) produces unspecified results. The form *symbol*=*value* is used by the KornShell *name*[*expression*]=*value* syntax.

Chapter 5

File Format Notation

The STDIN, STDOUT, STDERR, INPUT FILES, and OUTPUT FILES sections of the utility descriptions use a syntax to describe the data organization within the files, when that organization is not otherwise obvious. The syntax is similar to that used by the System Interfaces volume of IEEE Std 1003.1-200x *printf*( ) function, as described in this chapter. When used in STDIN or INPUT FILES sections of the utility descriptions, this syntax describes the format that could have been used to write the text to be read, not a format that could be used by the System Interfaces volume of IEEE Std 1003.1-200x *scanf*( ) function to read the input file.

The description of an individual record is as follows:

```
"<format>", [<arg1>, <arg2>,..., <argn>]
```

The *format* is a character string that contains three types of objects defined below:

1. *Characters* that are not *escape sequences* or *conversion specifications*, as described below, shall be copied to the output.
2. *Escape Sequences* represent non-graphic characters.
3. *Conversion Specifications* specify the output format of each argument; (see below).

The following characters have the following special meaning in the format string:

`' '` (An empty character position.) Represents one or more <blank>s.

Δ Represents exactly one <space>.

Table 5-1 lists escape sequences and associated actions on display devices capable of the action.

Table 5-1 Escape Sequences and Associated Actions

Escape Sequence	Represents Character	Terminal Action
`'\\'`	backslash	Print the character `'\'`.
`'\a'`	alert	Attempt to alert the user through audible or visible notification.
`'\b'`	backspace	Move the printing position to one column before the current position, unless the current position is the start of a line.
`'\f'`	form-feed	Move the printing position to the initial printing position of the next logical page.
`'\n'`	newline	Move the printing position to the start of the next line.
`'\r'`	carriage-return	Move the printing position to the start of the current line.
`'\t'`	tab	Move the printing position to the next tab position on the current line. If there are no more tab positions remaining on the line, the behavior is undefined.
`'\v'`	vertical-tab	Move the printing position to the start of the next vertical tab position. If there are no more vertical tab positions left on the page, the behavior is undefined.

Each conversion specification is introduced by the percent-sign character (`'%'`). After the character `'%'`, the following shall appear in sequence:

flags Zero or more *flags*, in any order, that modify the meaning of the conversion specification.

field width An optional string of decimal digits to specify a minimum *field width*. For an output field, if the converted value has fewer bytes than the field width, it shall be padded on the left (or right, if the left-adjustment flag (`'−'`), described below, has been given) to the field width.

precision Gives the minimum number of digits to appear for the `d`, `o`, `i`, `u`, `x`, or `X` conversion specifiers (the field is padded with leading zeros), the number of digits to appear after the radix character for the `e` and `f` conversion specifiers, the maximum number of significant digits for the `g` conversion specifier; or the maximum number of bytes to be written from a string in the `s` conversion specifier. The precision shall take the form of a period (`'.'`) followed by a decimal digit string; a null digit string is treated as zero.

conversion specifier characters
A conversion specifier character (see below) that indicates the type of conversion to be applied.

The *flag* characters and their meanings are:

– The result of the conversion shall be left-justified within the field.

\+ The result of a signed conversion shall always begin with a sign (`'+'` or `'−'`).

<space> If the first character of a signed conversion is not a sign, a <space> shall be prefixed to the result. This means that if the <space> and `'+'` flags both appear, the <space> flag shall be ignored.

The value shall be converted to an alternative form. For `c`, `d`, `i`, `u`, and `s` conversion specifiers, the behavior is undefined. For the `o` conversion specifier, it shall increase the precision to force the first digit of the result to be a zero. For `x` or `X` conversion specifiers, a non-zero result has 0x or 0X prefixed to it, respectively. For

`e`, `E`, `f`, `g`, and `G` conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. For `g` and `G` conversion specifiers, trailing zeros shall not be removed from the result as they usually are.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, and `G` conversion specifiers, leading zeros (following any indication of sign or base) shall be used to pad to the field width; no space padding is performed. If the `'0'` and `'–'` flags both appear, the `'0'` flag shall be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers, if a precision is specified, the `'0'` flag shall be ignored. For other conversion specifiers, the behavior is undefined.

Each conversion specifier character shall result in fetching zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be ignored.

The *conversion specifiers* and their meanings are:

`d,i,o,u,x,X` The integer argument shall be written as signed decimal (`d` or `i`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`). The `d` and `i` specifiers shall convert to signed decimal in the style `"[–]`*dddd*`"`. The `x` conversion specifier shall use the numbers and letters `"0123456789abcdef"` and the `X` conversion specifier shall use the numbers and letters `"0123456789ABCDEF"`. The *precision* component of the argument shall specify the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting a zero value with a precision of 0 shall be no characters. If both the field width and precision are omitted, the implementation may precede, follow, or precede and follow numeric arguments of types `d`, `i`, and `u` with <blank>s; arguments of type `o` (octal) may be preceded with leading zeros.

`f` The floating-point number argument shall be written in decimal notation in the style **[–]***ddd.ddd*, where the number of digits after the radix character (shown here as a decimal point) shall be equal to the *precision* specification. The *LC_NUMERIC* locale category shall determine the radix character to use in this format. If the *precision* is omitted from the argument, six digits shall be written after the radix character; if the *precision* is explicitly 0, no radix character shall appear.

`e,E` The floating-point number argument shall be written in the style **[–]***d.ddd*e±*dd* (the symbol `'±'` indicates either a plus or minus sign), where there is one digit before the radix character (shown here as a decimal point) and the number of digits after it is equal to the precision. The *LC_NUMERIC* locale category shall determine the radix character to use in this format. When the precision is missing, six digits shall be written after the radix character; if the precision is 0, no radix character shall appear. The `E` conversion specifier shall produce a number with `E` instead of `e` introducing the exponent. The exponent shall always contain at least two digits. However, if the value to be written requires an exponent greater than two digits, additional exponent digits shall be written as necessary.

`g,G` The floating-point number argument shall be written in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` (or `E`) shall be used only if the exponent resulting from the conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character shall appear only if it is followed by a digit.

`c`	The integer argument shall be converted to an **unsigned char** and the resulting byte shall be written.
`s`	The argument shall be taken to be a string and bytes from the string shall be written until the end of the string or the number of bytes indicated by the *precision* specification of the argument is reached. If the precision is omitted from the argument, it shall be taken to be infinite, so all bytes up to the end of the string shall be written.
`%`	Write a `'%'` character; no argument is converted.

In no case does a nonexistent or insufficient *field width* cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. The term *field width* should not be confused with the term *precision* used in the description of `%s`.

Examples

To represent the output of a program that prints a date and time in the form Sunday, July 3, 10:02, where *weekday* and *month* are strings:

```
"%s,Δ%sΔ%d,Δ%d:%.2d\n" <weekday>, <month>, <day>, <hour>, <min>
```

To show `'π'` written to 5 decimal places:

```
"piΔ=Δ%.5f\n",<value of π>
```

To show an input file format consisting of five colon-separated fields:

```
"%s:%s:%s:%s:%s\n", <arg1>, <arg2>, <arg3>, <arg4>, <arg5>
```

Chapter 6

Character Set

6.1 Portable Character Set

Conforming implementations shall support one or more coded character sets. Each supported locale shall include the *portable character set*, which is the set of symbolic names for characters in Table 6-1. This is used to describe characters within the text of IEEE Std 1003.1-200x. The first eight entries in Table 6-1 are defined in the ISO/IEC 6429:1992 standard and the rest of the characters are defined in the ISO/IEC 10646-1:2000 standard.

Table 6-1 Portable Character Set

Symbolic Name	Glyph	UCS	Description
<NUL>		<U0000>	NULL (NUL)
<alert>		<U0007>	BELL (BEL)
<backspace>		<U0008>	BACKSPACE (BS)
<tab>		<U0009>	CHARACTER TABULATION (HT)
<carriage-return>		<U000D>	CARRIAGE RETURN (CR)
<newline>		<U000A>	LINE FEED (LF)
<vertical-tab>		<U000B>	LINE TABULATION (VT)
<form-feed>		<U000C>	FORM FEED (FF)
<space>		<U0020>	SPACE
<exclamation-mark>	!	<U0021>	EXCLAMATION MARK
<quotation-mark>	"	<U0022>	QUOTATION MARK
<number-sign>	#	<U0023>	NUMBER SIGN
<dollar-sign>	$	<U0024>	DOLLAR SIGN
<percent-sign>	%	<U0025>	PERCENT SIGN
<ampersand>	&	<U0026>	AMPERSAND
<apostrophe>	’	<U0027>	APOSTROPHE
<left-parenthesis>	(	<U0028>	LEFT PARENTHESIS
<right-parenthesis>	)	<U0029>	RIGHT PARENTHESIS
<asterisk>	*	<U002A>	ASTERISK
<plus-sign>	+	<U002B>	PLUS SIGN
<comma>	,	<U002C>	COMMA
<hyphen-minus>	–	<U002D>	HYPHEN-MINUS
<hyphen>	-	<U002D>	HYPHEN-MINUS
<full-stop>	.	<U002E>	FULL STOP
<period>	.	<U002E>	FULL STOP
<slash>	/	<U002F>	SOLIDUS
<solidus>	/	<U002F>	SOLIDUS
<zero>	0	<U0030>	DIGIT ZERO
<one>	1	<U0031>	DIGIT ONE
<two>	2	<U0032>	DIGIT TWO
<three>	3	<U0033>	DIGIT THREE

Symbolic Name	Glyph	UCS	Description
<four>	4	<U0034>	DIGIT FOUR
<five>	5	<U0035>	DIGIT FIVE
<six>	6	<U0036>	DIGIT SIX
<seven>	7	<U0037>	DIGIT SEVEN
<eight>	8	<U0038>	DIGIT EIGHT
<nine>	9	<U0039>	DIGIT NINE
<colon>	:	<U003A>	COLON
<semicolon>	;	<U003B>	SEMICOLON
<less-than-sign>	<	<U003C>	LESS-THAN SIGN
<equals-sign>	=	<U003D>	EQUALS SIGN
<greater-than-sign>	>	<U003E>	GREATER-THAN SIGN
<question-mark>	?	<U003F>	QUESTION MARK
<commercial-at>	@		<U0040>
<A>	A	<U0041>	LATIN CAPITAL LETTER A
<B>	B	<U0042>	LATIN CAPITAL LETTER B
<C>	C	<U0043>	LATIN CAPITAL LETTER C
<D>	D	<U0044>	LATIN CAPITAL LETTER D
<E>	E	<U0045>	LATIN CAPITAL LETTER E
<F>	F	<U0046>	LATIN CAPITAL LETTER F
<G>	G	<U0047>	LATIN CAPITAL LETTER G
<H>	H	<U0048>	LATIN CAPITAL LETTER H
<I>	I	<U0049>	LATIN CAPITAL LETTER I
<J>	J	<U004A>	LATIN CAPITAL LETTER J
<K>	K	<U004B>	LATIN CAPITAL LETTER K
<L>	L	<U004C>	LATIN CAPITAL LETTER L
<M>	M	<U004D>	LATIN CAPITAL LETTER M
<N>	N	<U004E>	LATIN CAPITAL LETTER N
<O>	O	<U004F>	LATIN CAPITAL LETTER O
<P>	P	<U0050>	LATIN CAPITAL LETTER P
<Q>	Q	<U0051>	LATIN CAPITAL LETTER Q
<R>	R	<U0052>	LATIN CAPITAL LETTER R
<S>	S	<U0053>	LATIN CAPITAL LETTER S
<T>	T	<U0054>	LATIN CAPITAL LETTER T
<U>	U	<U0055>	LATIN CAPITAL LETTER U
<V>	V	<U0056>	LATIN CAPITAL LETTER V
<W>	W	<U0057>	LATIN CAPITAL LETTER W
<X>	X	<U0058>	LATIN CAPITAL LETTER X
<Y>	Y	<U0059>	LATIN CAPITAL LETTER Y
<Z>	Z	<U005A>	LATIN CAPITAL LETTER Z
<left-square-bracket>	[	<U005B>	LEFT SQUARE BRACKET
<backslash>	\	<U005C>	REVERSE SOLIDUS
<reverse-solidus>	\	<U005C>	REVERSE SOLIDUS
<right-square-bracket>	]	<U005D>	RIGHT SQUARE BRACKET
<circumflex-accent>	^	<U005E>	CIRCUMFLEX ACCENT
<circumflex>	^	<U005E>	CIRCUMFLEX ACCENT
<low-line>	_	<U005F>	LOW LINE
<underscore>	_	<U005F>	LOW LINE

Symbolic Name	Glyph	UCS	Description
<grave-accent>	‘	<U0060>	GRAVE ACCENT
<a>	a	<U0061>	LATIN SMALL LETTER A
<b>	b	<U0062>	LATIN SMALL LETTER B
<c>	c	<U0063>	LATIN SMALL LETTER C
<d>	d	<U0064>	LATIN SMALL LETTER D
<e>	e	<U0065>	LATIN SMALL LETTER E
<f>	f	<U0066>	LATIN SMALL LETTER F
<g>	g	<U0067>	LATIN SMALL LETTER G
<h>	h	<U0068>	LATIN SMALL LETTER H
<i>	i	<U0069>	LATIN SMALL LETTER I
<j>	j	<U006A>	LATIN SMALL LETTER J
<k>	k	<U006B>	LATIN SMALL LETTER K
<l>	l	<U006C>	LATIN SMALL LETTER L
<m>	m	<U006D>	LATIN SMALL LETTER M
<n>	n	<U006E>	LATIN SMALL LETTER N
<o>	o	<U006F>	LATIN SMALL LETTER O
<p>	p	<U0070>	LATIN SMALL LETTER P
<q>	q	<U0071>	LATIN SMALL LETTER Q
<r>	r	<U0072>	LATIN SMALL LETTER R
<s>	s	<U0073>	LATIN SMALL LETTER S
<t>	t	<U0074>	LATIN SMALL LETTER T
<u>	u	<U0075>	LATIN SMALL LETTER U
<v>	v	<U0076>	LATIN SMALL LETTER V
<w>	w	<U0077>	LATIN SMALL LETTER W
<x>	x	<U0078>	LATIN SMALL LETTER X
<y>	y	<U0079>	LATIN SMALL LETTER Y
<z>	z	<U007A>	LATIN SMALL LETTER Z
<left-brace>	{	<U007B>	LEFT CURLY BRACKET
<left-curly-bracket>	{	<U007B>	LEFT CURLY BRACKET
<vertical-line>	|	<U007C>	VERTICAL LINE
<right-brace>	}	<U007D>	RIGHT CURLY BRACKET
<right-curly-bracket>	}	<U007D>	RIGHT CURLY BRACKET
<tilde>	~	<U007E>	TILDE

IEEE Std 1003.1-200x uses character names other than the above, but only in an informative way; for example, in examples to illustrate the use of characters beyond the portable character set with the facilities of IEEE Std 1003.1-200x.

Table 6-1 (on page 111) defines the characters in the portable character set and the corresponding symbolic character names used to identify each character in a character set description file. The table contains more than one symbolic character name for characters whose traditional name differs from the chosen name. Characters defined in Table 6-2 (on page 116) may also be used in character set description files.

IEEE Std 1003.1-200x places only the following requirements on the encoded values of the characters in the portable character set:

- If the encoded values associated with each member of the portable character set are not invariant across all locales supported by the implementation, if an application accesses any pair of locales where the character encodings differ, or accesses data from an application running in a locale which has different encodings from the application's current locale, the results are unspecified.

- The encoded values associated with the digits 0 to 9 shall be such that the value of each character after 0 shall be one greater than the value of the previous character.
- A null character, NUL, which has all bits set to zero, shall be in the set of characters.
- The encoded values associated with the members of the portable character set are each represented in a single byte. Moreover, if the value is stored in an object of C-language type **char**, it is guaranteed to be positive (except the NUL, which is always zero).

Conforming implementations shall support certain character and character set attributes, as defined in Section 7.2 (on page 120).

6.2 Character Encoding

The POSIX locale contains the characters in Table 6-1 (on page 111), which have the properties listed in Section 7.3.1 (on page 122). In other locales, the presence, meaning, and representation of any additional characters is locale-specific.

In locales other than the POSIX locale, a character may have a state-dependent encoding. There are two types of these encodings:

- A single-shift encoding (where each character not in the initial shift state is preceded by a shift code) can be defined if each shift-code and character sequence is considered a multi-byte character. This is done using the concatenated-constant format in a character set description file, as described in Section 6.4 (on page 115). If the implementation supports a character encoding of this type, all of the standard utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x shall support it. Use of a single-shift encoding with any of the functions in the System Interfaces volume of IEEE Std 1003.1-200x that do not specifically mention the effects of state-dependent encoding is implementation-defined.
- A locking-shift encoding (where the state of the character is determined by a shift code that may affect more than the single character following it) cannot be defined with the current character set description file format. Use of a locking-shift encoding with any of the standard utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x or with any of the functions in the System Interfaces volume of IEEE Std 1003.1-200x that do not specifically mention the effects of state-dependent encoding is implementation-defined.

While in the initial shift state, all characters in the portable character set shall retain their usual interpretation and shall not alter the shift state. The interpretation for subsequent bytes in the sequence shall be a function of the current shift state. A byte with all bits zero shall be interpreted as the null character independent of shift state. Thus a byte with all bits zero shall never occur in the second or subsequent bytes of a character.

The maximum allowable number of bytes in a character in the current locale shall be indicated by {MB_CUR_MAX}, defined in the **<stdlib.h>** header and by the **<mb_cur_max>** value in a character set description file; see Section 6.4 (on page 115). The implementation's maximum number of bytes in a character shall be defined by the C-language macro {MB_LEN_MAX}.

6.3 C Language Wide-Character Codes

In the shell, the standard utilities are written so that the encodings of characters are described by the locale's *LC_CTYPE* definition (see Section 7.3.1 (on page 122)) and there is no differentiation between characters consisting of single octets (8-bit bytes) or multiple bytes. However, in the C language, a differentiation is made. To ease the handling of variable length characters, the C language has introduced the concept of wide-character codes.

All wide-character codes in a given process consist of an equal number of bits. This is in contrast to characters, which can consist of a variable number of bytes. The byte or byte sequence that represents a character can also be represented as a wide-character code. Wide-character codes thus provide a uniform size for manipulating text data. A wide-character code having all bits zero is the null wide-character code (see Section 3.246 (on page 66)), and terminates wide-character strings (see Section 3.432 (on page 92)). The wide-character value for each member of the portable character set shall equal its value when used as the lone character in an integer character constant. Wide-character codes for other characters are locale and implementation-defined. State shift bytes shall not have a wide-character code representation.

6.4 Character Set Description File

Implementations shall provide a character set description file for at least one coded character set supported by the implementation. These files are referred to elsewhere in IEEE Std 1003.1-200x as *charmap* files. It is implementation-defined whether or not users or applications can provide additional character set description files.

IEEE Std 1003.1-200x does not require that multiple character sets or codesets be supported. Although multiple charmap files are supported, it is the responsibility of the implementation to provide the file or files; if only one is provided, only that one is accessible using the *localedef* utility's −**f** option.

Each character set description file, except those that use the ISO/IEC 10646-1:2000 standard position values as the encoding values, shall define characteristics for the coded character set and the encoding for the characters specified in Table 6-1 (on page 111), and may define encoding for additional characters supported by the implementation. Other information about the coded character set may also be in the file. Coded character set character values shall be defined using symbolic character names followed by character encoding values.

Each symbolic name specified in Table 6-1 (on page 111) shall be included in the file and shall be mapped to a unique coding value, except as noted below. The glyphs `'{'`, `'}'`, `'_'`, `'-'`, `'/'`, `'\'`, `'.'`, and `'^'` have more than one symbolic name; all symbolic names for each such glyph shall be included, each with identical encoding. If some or all of the control characters identified in Table 6-2 (on page 116) are supported by the implementation, the symbolic names and their corresponding encoding values shall be included in the file. Some of the encodings associated with the symbolic names in Table 6-2 (on page 116) may be the same as characters found in Table 6-1 (on page 111); both names shall be provided for each encoding.

Table 6-2 Control Character Set

<ACK>	<DC2>	<ENQ>	<FS>	<IS4>	<SOH>
<BEL>	<DC3>	<EOT>	<GS>	<LF>	<STX>
<BS>	<DC4>	<ESC>	<HT>	<NAK>	<SUB>
<CAN>	<DEL>	<ETB>	<IS1>	<RS>	<SYN>
<CR>	<DLE>	<ETX>	<IS2>	<SI>	<US>
<DC1>	<EM>	<FF>	<IS3>	<SO>	<VT>

The following declarations can precede the character definitions. Each shall consist of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more <blank>s, followed by the value to be assigned to the symbol.

<code_set_name>	The name of the coded character set for which the character set description file is defined. The characters of the name shall be taken from the set of characters with visible glyphs defined in Table 6-1 (on page 111).
<mb_cur_max>	The maximum number of bytes in a multi-byte character. This shall default to 1.
<mb_cur_min>	An unsigned positive integer value that defines the minimum number of bytes in a character for the encoded character set. XSI On XSI-conformant systems, **<mb_cur_min>** shall always be 1.
<escape_char>	The character used to indicate that the characters following shall be interpreted in a special way, as defined later in this section. This shall default to backslash (`'\'`), which is the character used in all the following text and examples, unless otherwise noted.
<comment_char>	The character that, when placed in column 1 of a charmap line, is used to indicate that the line shall be ignored. The default character shall be the number sign (`'#'`).

The character set mapping definitions shall be all the lines immediately following an identifier line containing the string `"CHARMAP"` starting in column 1, and preceding a trailer line containing the string `"END CHARMAP"` starting in column 1. Empty lines and lines containing a **<comment_char>** in the first column shall be ignored. Each non-comment line of the character set mapping definition (that is, between the `"CHARMAP"` and `"END CHARMAP"` lines of the file) shall be in either of two forms:

```
"%s %s %s\n", <symbolic-name>, <encoding>, <comments>
```

or:

```
"%s...%s %s %s\n", <symbolic-name>, <symbolic-name>,
    <encoding>, <comments>
```

In the first format, the line in the character set mapping definition shall define a single symbolic name and a corresponding encoding. A symbolic name is one or more characters from the set shown with visible glyphs in Table 6-1 (on page 111), enclosed between angle brackets. A character following an escape character is interpreted as itself; for example, the sequence `"<\\\>>"` represents the symbolic name `"\>"` enclosed between angle brackets.

In the second format, the line in the character set mapping definition shall define a range of one or more symbolic names. In this form, the symbolic names shall consist of zero or more non-numeric characters from the set shown with visible glyphs in Table 6-1 (on page 111), followed by an integer formed by one or more decimal digits. Both integers shall contain the same number of digits. The characters preceding the integer shall be identical in the two symbolic names, and the integer formed by the digits in the second symbolic name shall be equal to or greater than the integer formed by the digits in the first name. This shall be interpreted as a series of symbolic names formed from the common part and each of the integers between the first and the second integer, inclusive. As an example, <j0101>...<j0104> is interpreted as the symbolic names <j0101>, <j0102>, <j0103>, and <j0104>, in that order.

A character set mapping definition line shall exist for all symbolic names specified in Table 6-1 (on page 111), and shall define the coded character value that corresponds to the character indicated in the table, or the coded character value that corresponds to the control character symbolic name. If the control characters commonly associated with the symbolic names in Table 6-2 (on page 116) are supported by the implementation, the symbolic name and the corresponding encoding value shall be included in the file. Additional unique symbolic names may be included. A coded character value can be represented by more than one symbolic name.

The encoding part is expressed as one (for single-byte character values) or more concatenated decimal, octal, or hexadecimal constants in the following formats:

```
"%cd%u", <escape_char>, <decimal byte value>
"%cx%x", <escape_char>, <hexadecimal byte value>
"%c%o", <escape_char>, <octal byte value>
```

Decimal constants shall be represented by two or three decimal digits, preceded by the escape character and the lowercase letter `'d'`; for example, `"\d05"`, `"\d97"`, or `"\d143"`. Hexadecimal constants shall be represented by two hexadecimal digits, preceded by the escape character and the lowercase letter `'x'`; for example, `"\x05"`, `"\x61"`, or `"\x8f"`. Octal constants shall be represented by two or three octal digits, preceded by the escape character; for example, `"\05"`, `"\141"`, or `"\217"`. In a portable charmap file, each constant represents an 8-bit byte. When constants are concatenated for multi-byte character values, they shall be of the same type, and interpreted in byte order from first to last with the least significant byte of the multi-byte character specified by the last constant. The manner in which these constants are represented in the character stored in the system is implementation-defined. (This notation was chosen for reasons of portability. There is no requirement that the internal representation in the computer memory be in this same order.) Omitting bytes from a multi-byte character definition produces undefined results.

In lines defining ranges of symbolic names, the encoded value shall be the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent symbolic names defined by the range shall have encoding values in increasing order. Bytes shall be treated as unsigned octets, and carry shall be propagated between the bytes as necessary to represent the range. For example, the line:

```
<j0101>...<j0104>  \d129\d254
```

is interpreted as:

```
<j0101>            \d129\d254
<j0102>            \d129\d255
<j0103>            \d130\d0
<j0104>            \d130\d1
```

Note that this line is interpreted as the example even on systems with bytes larger than 8 bits.

The comment is optional.

The following declarations can follow the character set mapping definitions (after the `"END CHARMAP"` statement). Each shall consist of the keyword shown in the following list, starting in column 1, followed by the value(s) to be associated to the keyword, as defined below.

WIDTH	An unsigned positive integer value defining the column width (see Section 3.103 (on page 47)) for the printable characters in the coded character set specified in Table 6-1 (on page 111) and Table 6-2 (on page 116). Coded character set character values shall be defined using symbolic character names followed by column width values. Defining a character with more than one **WIDTH** produces undefined results. The **END WIDTH** keyword shall be used to terminate the **WIDTH** definitions. Specifying the width of a non-printable character in a **WIDTH** declaration produces undefined results.
WIDTH_DEFAULT	An unsigned positive integer value defining the default column width for any printable character not listed by one of the **WIDTH** keywords. If no **WIDTH_DEFAULT** keyword is included in the charmap, the default character width shall be 1.

Example

After the `"END CHARMAP"` statement, a syntax for a width definition would be:

```
WIDTH
<A> 1
<B> 1
<C>...<Z> 1
<foo1>...<foon> 2
END WIDTH
```

In this example, the numerical code point values represented by the symbols **<A>** and **<B>** are assigned a width of 1. The code point values **<C>** to **<Z>** inclusive (**<C>**, **<D>**, **<E>**, and so on) are also assigned a width of 1. Using **<A>**...**<Z>** would have required fewer lines, but the alternative was shown to demonstrate flexibility. The keyword **WIDTH_DEFAULT** could have been added as appropriate.

6.4.1 State-Dependent Character Encodings

This section addresses the use of state-dependent character encodings (that is, those in which the encoding of a character is dependent on one or more shift codes that may precede it).

A single-shift encoding (where each character not in the initial shift state is preceded by a shift code) can be defined in the charmap format if each shift-code/character sequence is considered a multi-byte character, defined using the concatenated-constant format described in Section 6.4 (on page 115). If the implementation supports a character encoding of this type, all of the standard utilities shall support it. A locking-shift encoding (where the state of the character is determined by a shift code that may affect more than the single character following it) could be defined with an extension to the charmap format described in Section 6.4 (on page 115). If the implementation supports a character encoding of this type, any of the standard utilities that describe character (*versus* byte) or text-file manipulation shall have the following characteristics:

1. The utility shall process the statefully encoded data as a concatenation of state-independent characters. The presence of redundant locking shifts shall not affect the comparison of two statefully encoded strings.

2. A utility that divides, truncates, or extracts substrings from statefully encoded data shall produce output that contains locking shifts at the beginning or end of the resulting data, if appropriate, to retain correct state information.

Chapter 7

Locale

7.1 General

A *locale* is the definition of the subset of a user's environment that depends on language and cultural conventions. It is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behavior of components of the system. Category names correspond to the following environment variable names:

LC_CTYPE	Character classification and case conversion.
LC_COLLATE	Collation order.
LC_MONETARY	Monetary formatting.
LC_NUMERIC	Numeric, non-monetary formatting.
LC_TIME	Date and time formats.
LC_MESSAGES	Formats of informative and diagnostic messages and interactive responses.

The standard utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x shall base their behavior on the current locale, as defined in the ENVIRONMENT VARIABLES section for each utility. The behavior of some of the C-language functions defined in the System Interfaces volume of IEEE Std 1003.1-200x shall also be modified based on the current locale, as defined by the last call to *setlocale*( ).

Locales other than those supplied by the implementation can be created via the *localedef* utility, provided that the _POSIX2_LOCALEDEF symbol is defined on the system. Even if *localedef* is not provided, all implementations conforming to the System Interfaces volume of IEEE Std 1003.1-200x shall provide one or more locales that behave as described in this chapter. The input to the utility is described in Section 7.3 (on page 120). The value that is used to specify a locale when using environment variables shall be the string specified as the *name* operand to the *localedef* utility when the locale was created. The strings `"C"` and `"POSIX"` are reserved as identifiers for the POSIX locale (see Section 7.2 (on page 120)). When the value of a locale environment variable begins with a slash (`'/'`), it shall be interpreted as the pathname of the locale definition; the type of file (regular, directory, and so on) used to store the locale definition is implementation-defined. If the value does not begin with a slash, the mechanism used to locate the locale is implementation-defined.

If different character sets are used by the locale categories, the results achieved by an application utilizing these categories are undefined. Likewise, if different codesets are used for the data being processed by interfaces whose behavior is dependent on the current locale, or the codeset is different from the codeset assumed when the locale was created, the result is also undefined.

Applications can select the desired locale by invoking the *setlocale*( ) function (or equivalent) with the appropriate value. If the function is invoked with an empty string, such as:

```
setlocale(LC_ALL, "");
```

the value of the corresponding environment variable is used. If the environment variable is unset or is set to the empty string, the implementation shall set the appropriate environment as defined in Chapter 8 (on page 157).

7.2 POSIX Locale

Conforming systems shall provide a *POSIX locale*, also known as the C locale. The behavior of standard utilities and functions in the POSIX locale shall be as if the locale was defined via the *localedef* utility with input data from the POSIX locale tables in Section 7.3.

The tables in Section 7.3 describe the characteristics and behavior of the POSIX locale for data consisting entirely of characters from the portable character set and the control character set. For other characters, the behavior is unspecified. For C-language programs, the POSIX locale shall be the default locale when the *setlocale*( ) function is not called.

The POSIX locale can be specified by assigning to the appropriate environment variables the values `"C"` or `"POSIX"`.

All implementations shall define a locale as the default locale, to be invoked when no environment variables are set, or set to the empty string. This default locale can be the POSIX locale or any other implementation-defined locale. Some implementations may provide facilities for local installation administrators to set the default locale, customizing it for each location. IEEE Std 1003.1-200x does not require such a facility.

7.3 Locale Definition

The capability to specify additional locales to those provided by an implementation is optional, denoted by the _POSIX2_LOCALEDEF symbol. If the option is not supported, only implementation-supplied locales are available. Such locales shall be documented using the format specified in this section.

Locales can be described with the file format presented in this section. The file format is that accepted by the *localedef* utility. For the purposes of this section, the file is referred to as the *locale definition file*, but no locales shall be affected by this file unless it is processed by *localedef* or some similar mechanism. Any requirements in this section imposed upon the utility shall apply to *localedef* or to any other similar utility used to install locale information using the locale definition file format described here.

The locale definition file shall contain one or more locale category source definitions, and shall not contain more than one definition for the same locale category. If the file contains source definitions for more than one category, implementation-defined categories, if present, shall appear after the categories defined by Section 7.1 (on page 119). A category source definition contains either the definition of a category or a **copy** directive. For a description of the **copy** directive, see *localedef*. In the event that some of the information for a locale category, as specified in this volume of IEEE Std 1003.1-200x, is missing from the locale source definition, the behavior of that category, if it is referenced, is unspecified.

A category source definition shall consist of a category header, a category body, and a category trailer. A category header shall consist of the character string naming of the category, beginning with the characters *LC_*. The category trailer shall consist of the string `"END"`, followed by one or more <blank>s and the string used in the corresponding category header.

The category body shall consist of one or more lines of text. Each line shall contain an identifier, optionally followed by one or more operands. Identifiers shall be either keywords, identifying a particular locale element, or collating elements. In addition to the keywords defined in this volume of IEEE Std 1003.1-200x, the source can contain implementation-defined keywords. Each keyword within a locale shall have a unique name (that is, two categories cannot have a commonly-named keyword); no keyword shall start with the characters *LC_*. Identifiers shall be separated from the operands by one or more <blank>s.

Operands shall be characters, collating elements, or strings of characters. Strings shall be enclosed in double-quotes. Literal double-quotes within strings shall be preceded by the <*escape character*>, described below. When a keyword is followed by more than one operand, the operands shall be separated by semicolons; <blank>s shall be allowed both before and after a semicolon.

The first category header in the file can be preceded by a line modifying the comment character. It shall have the following format, starting in column 1:

```
"comment_char %c\n", <comment character>
```

The comment character shall default to the number sign (`'#'`). Blank lines and lines containing the <*comment character*> in the first position shall be ignored.

The first category header in the file can be preceded by a line modifying the escape character to be used in the file. It shall have the following format, starting in column 1:

```
"escape_char %c\n", <escape character>
```

The escape character shall default to backslash, which is the character used in all examples shown in this volume of IEEE Std 1003.1-200x.

A line can be continued by placing an escape character as the last character on the line; this continuation character shall be discarded from the input. Although the implementation need not accept any one portion of a continued line with a length exceeding {LINE_MAX} bytes, it shall place no limits on the accumulated length of the continued line. Comment lines shall not be continued on a subsequent line using an escaped newline character.

Individual characters, characters in strings, and collating elements shall be represented using symbolic names, as defined below. In addition, characters can be represented using the characters themselves or as octal, hexadecimal, or decimal constants. When non-symbolic notation is used, the resultant locale definitions are in many cases not portable between systems. The left angle bracket (`'<'`) is a reserved symbol, denoting the start of a symbolic name; when used to represent itself it shall be preceded by the escape character. The following rules apply to character representation:

1. A character can be represented via a symbolic name, enclosed within angle brackets `'<'` and `'>'`. The symbolic name, including the angle brackets, shall exactly match a symbolic name defined in the charmap file specified via the *localedef* −**f** option, and it shall be replaced by a character value determined from the value associated with the symbolic name in the charmap file. The use of a symbolic name not found in the charmap file shall constitute an error, unless the category is *LC_CTYPE* or *LC_COLLATE*, in which case it shall constitute a warning condition (see *localedef* for a description of actions resulting from errors and warnings). The specification of a symbolic name in a **collating-element** or **collating-symbol** section that duplicates a symbolic name in the charmap file (if present) shall be an error. Use of the escape character or a right angle bracket within a symbolic name is invalid unless the character is preceded by the escape character.

   For example:

   ```
   <c>;<c-cedilla>  "<M><a><y>"
   ```

2. A character in the portable character set can be represented by the character itself, in which case the value of the character is implementation-defined. (Implementations may allow other characters to be represented as themselves, but such locale definitions are not portable.) Within a string, the double-quote character, the escape character, and the right angle bracket character shall be escaped (preceded by the escape character) to be interpreted as the character itself. Outside strings, the characters:

```
, ; < > escape_char
```

shall be escaped to be interpreted as the character itself.

For example:

```
c "May"
```

3. A character can be represented as an octal constant. An octal constant shall be specified as the escape character followed by two or three octal digits. Each constant shall represent a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   For example:

   ```
   \143;\347;\143\150   "\115\141\171"
   ```

4. A character can be represented as a hexadecimal constant. A hexadecimal constant shall be specified as the escape character followed by an `'x'` followed by two hexadecimal digits. Each constant shall represent a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   For example:

   ```
   \x63;\xe7;\x63\x68   "\x4d\x61\x79"
   ```

5. A character can be represented as a decimal constant. A decimal constant shall be specified as the escape character followed by a `'d'` followed by two or three decimal digits. Each constant represents a byte value. Multi-byte values can be represented by concatenated constants specified in byte order with the last constant specifying the least significant byte of the character.

   For example:

   ```
   \d99;\d231;\d99\d104  "\d77\d97\d121"
   ```

Implementations may accept single-digit octal, decimal, or hexadecimal constants following the escape character. Only characters existing in the character set for which the locale definition is created shall be specified, whether using symbolic names, the characters themselves, or octal, decimal, or hexadecimal constants. If a charmap file is present, only characters defined in the charmap can be specified using octal, decimal, or hexadecimal constants. Symbolic names not present in the charmap file can be specified and shall be ignored, as specified under item 1 above. |

7.3.1 LC_CTYPE

The *LC_CTYPE* category shall define character classification, case conversion, and other character attributes. In addition, a series of characters can be represented by three adjacent periods representing an ellipsis symbol (`"..."`). The ellipsis specification shall be interpreted as meaning that all values between the values preceding and following it represent valid characters. The ellipsis specification shall be valid only within a single encoded character set; that is, within a group of characters of the same size. An ellipsis shall be interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

For example:

```
\x30;...;\x39;
```

includes in the character class all characters with encoded values between the endpoints.

The following keywords shall be recognized. In the descriptions, the term ‘‘automatically included’’ means that it shall not be an error either to include or omit any of the referenced characters; the implementation provides them if missing (even if the entire keyword is missing) and accepts them silently if present. When the implementation automatically includes a missing character, it shall have an encoded value dependent on the charmap file in effect (see the description of the *localedef* **–f** option); otherwise, it shall have a value derived from an implementation-defined character mapping.

The character classes **digit**, **xdigit**, **lower**, **upper**, and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differ from the implementation default values. It is not possible to define a locale without these automatically included characters unless some implementation extension is used to prevent their inclusion. Such a definition would not be a proper superset of the C or POSIX locale and thus, it might not be possible for conforming applications to work properly.

copy Specify the name of an existing locale which shall be used as the definition of this category. If this keyword is specified, no other keyword shall be specified.

upper Define characters to be classified as uppercase letters.

In the POSIX locale, the 26 uppercase letters shall be included:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** shall be specified. The uppercase letters <A> to <Z>, as defined in Section 6.4 (on page 115) (the portable character set), are automatically included in this class.

lower Define characters to be classified as lowercase letters.

In the POSIX locale, the 26 lowercase letters shall be included:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** shall be specified. The lowercase letters <a> to <z> of the portable character set are automatically included in this class.

alpha Define characters to be classified as letters.

In the POSIX locale, all characters in the classes **upper** and **lower** shall be included.

In a locale definition file, no character specified for the keywords **cntrl**, **digit**, **punct**, or **space** shall be specified. Characters classified as either **upper** or **lower** are automatically included in this class.

digit Define the characters to be classified as numeric digits.

In the POSIX locale, only:

`0 1 2 3 4 5 6 7 8 9`

shall be included.

In a locale definition file, only the digits <zero>, <one>, <two>, <three>, <four>, <five>, <six>, <seven>, <eight>, and <nine> shall be specified, and in contiguous ascending sequence by numerical value. The digits <zero> to <nine> of the portable character set are automatically included in this class.

alnum Define characters to be classified as letters and numeric digits. Only the characters specified for the **alpha** and **digit** keywords shall be specified. Characters specified for the keywords **alpha** and **digit** are automatically included in this class.

space Define characters to be classified as white-space characters.

In the POSIX locale, at a minimum, the <space>, <form-feed>, <newline>, <carriage-return>, <tab>, and <vertical-tab> shall be included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **graph**, or **xdigit** shall be specified. The <space>, <form-feed>, <newline>, <carriage-return>, <tab>, and <vertical-tab> of the portable character set, and any characters included in the class **blank** are automatically included in this class.

cntrl Define characters to be classified as control characters.

In the POSIX locale, no characters in classes **alpha** or **print** shall be included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **punct**, **graph**, **print**, or **xdigit** shall be specified.

punct Define characters to be classified as punctuation characters.

In the POSIX locale, neither the <space> nor any characters in classes **alpha**, **digit**, or **cntrl** shall be included.

In a locale definition file, no character specified for the keywords **upper**, **lower**, **alpha**, **digit**, **cntrl**, **xdigit**, or as the <space> shall be specified.

graph Define characters to be classified as printable characters, not including the <space>.

In the POSIX locale, all characters in classes **alpha**, **digit**, and **punct** shall be included; no characters in class **cntrl** shall be included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, and **punct** are automatically included in this class. No character specified for the keyword **cntrl** shall be specified.

print Define characters to be classified as printable characters, including the <space>.

In the POSIX locale, all characters in class **graph** shall be included; no characters in class **cntrl** shall be included.

In a locale definition file, characters specified for the keywords **upper**, **lower**, **alpha**, **digit**, **xdigit**, **punct, graph**, and the <space> are automatically included in this class. No character specified for the keyword **cntrl** shall be specified.

xdigit Define the characters to be classified as hexadecimal digits.

In the POSIX locale, only:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
```

shall be included.

In a locale definition file, only the characters defined for the class **digit** shall be specified, in contiguous ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 to 15 inclusive, with each set in ascending order (for example, <A>, <B>, <C>, <D>, <E>, <F>, <a>, <b>, <c>, <d>, <e>, <f>). The digits <zero> to <nine>, the uppercase letters <A> to <F>, and the lowercase letters <a> to <f> of the portable character set are automatically included in this class.

blank Define characters to be classified as <blank>s.

In the POSIX locale, only the <space> and <tab> shall be included.

In a locale definition file, the <space> and <tab> are automatically included in this class. LI **charclass** Define one or more locale-specific character class names as strings separated by semicolons. Each named character class can then be defined subsequently in the *LC_CTYPE* definition. A character class name shall consist of at least one and at most {CHARCLASS_NAME_MAX} bytes of alphanumeric characters from the portable filename character set. The first character of a character class name shall not be a digit. The name shall not match any of the *LC_CTYPE* keywords defined in this volume of IEEE Std 1003.1-200x. Future revisions of IEEE Std 1003.1-200x will not specify any *LC_CTYPE* keywords containing uppercase letters.

charclass-name Define characters to be classified as belonging to the named locale-specific character class. In the POSIX locale, locale-specific named character classes need not exist.

If a class name is defined by a **charclass** keyword, but no characters are subsequently assigned to it, this is not an error; it represents a class without any characters belonging to it.

The *charclass-name* can be used as the *property* argument to the *wctype*() function, in regular expression and shell pattern-matching bracket expressions, and by the *tr* command.

toupper Define the mapping of lowercase letters to uppercase letters.

In the POSIX locale, at a minimum, the 26 lowercase characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

shall be mapped to the corresponding 26 uppercase characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

In a locale definition file, the operand shall consist of character pairs, separated by semicolons. The characters in each character pair shall be separated by a comma and the pair enclosed by parentheses. The first character in each pair is the lowercase letter, the second the corresponding uppercase letter. Only characters specified for the keywords **lower** and **upper** shall be specified. The lowercase letters <a> to <z>, and their corresponding uppercase letters <A> to <Z>, of the portable character set are automatically included in this mapping, but only when the **toupper** keyword is omitted from the locale definition.

tolower Define the mapping of uppercase letters to lowercase letters.

In the POSIX locale, at a minimum, the 26 uppercase characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

shall be mapped to the corresponding 26 lowercase characters:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

In a locale definition file, the operand shall consist of character pairs, separated by semicolons. The characters in each character pair shall be separated by a comma and the pair enclosed by parentheses. The first character in each pair is the uppercase letter, the second the corresponding lowercase letter. Only characters specified for the keywords **lower** and **upper** shall be specified. If the **tolower** keyword is omitted from the locale definition, the mapping is the reverse mapping of the one specified for **toupper**.

The following table shows the character class combinations allowed:

Table 7-1 Valid Character Class Combinations

	Can Also Belong To										
In Class	**upper**	**lower**	**alpha**	**digit**	**space**	**cntrl**	**punct**	**graph**	**print**	**xdigit**	**blank**
upper		—	A	x	x	x	x	A	A	—	x
lower	—		A	x	x	x	x	A	A	—	x
alpha	—	—		x	x	x	x	A	A	—	x
digit	x	x	x		x	x	x	A	A	A	x
space	x	x	x	x		—	*	*	*	x	—
cntrl	x	x	x	x	—		x	x	x	x	—
punct	x	x	x	x	—	x		A	A	x	—
graph	—	—	—	—	—	x	—		A	—	—
print	—	—	—	—	—	x	—	—		—	—
xdigit	—	—	—	—	x	x	x	A	A		x
blank	x	x	x	x	A	—	*	*	*	x	

Notes:

1. Explanation of codes:

   A Automatically included; see text.

   — Permitted.

   x Mutually-exclusive.

   * See note 2.

2. The <space>, which is part of the **space** and **blank** classes, cannot belong to **punct** or **graph**, but shall automatically belong to the **print** class. Other **space** or **blank** characters can be classified as any of **punct**, **graph**, or **print**.

7.3.1.1 *LC_CTYPE Category in the POSIX Locale*

The character classifications for the POSIX locale follow; the code listing depicts the *localedef* input, the table represents the same information, sorted by character.

```
LC_CTYPE
# The following is the POSIX locale LC_CTYPE.
# "alpha" is by default "upper" and "lower"
# "alnum" is by definition "alpha" and "digit"
# "print" is by default "alnum", "punct" and the <space>
# "graph" is by default "alnum" and "punct"
#
upper    <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;<L>;<M>;\
         <N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;<X>;<Y>;<Z>
#
```

```
lower    <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;<l>;<m>;\
         <n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;<x>;<y>;<z>
#
digit    <zero>;<one>;<two>;<three>;<four>;<five>;<six>;\
         <seven>;<eight>;<nine>
#
space    <tab>;<newline>;<vertical-tab>;<form-feed>;\
         <carriage-return>;<space>
#
cntrl    <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
         <form-feed>;<carriage-return>;\
         <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
         <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
         <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
         <IS1>;<DEL>
#
punct    <exclamation-mark>;<quotation-mark>;<number-sign>;\
         <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
         <left-parenthesis>;<right-parenthesis>;<asterisk>;\
         <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
         <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
         <greater-than-sign>;<question-mark>;<commercial-at>;\
         <left-square-bracket>;<backslash>;<right-square-bracket>;\
         <circumflex>;<underscore>;<grave-accent>;<left-curly-bracket>;\
         <vertical-line>;<right-curly-bracket>;<tilde>
#
xdigit   <zero>;<one>;<two>;<three>;<four>;<five>;<six>;<seven>;\
         <eight>;<nine>;<A>;<B>;<C>;<D>;<E>;<F>;<a>;<b>;<c>;<d>;<e>;<f>
#
blank    <space>;<tab>
#
toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);(<z>,<Z>)
#
tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);(<Z>,<z>)
END LC_CTYPE
```

Symbolic Name	Other Case	Character Classes
<NUL>		**cntrl**
<SOH>		**cntrl**
<STX>		**cntrl**
<ETX>		**cntrl**
<EOT>		**cntrl**
<ENQ>		**cntrl**
<ACK>		**cntrl**
<alert>		**cntrl**
<backspace>		**cntrl**
<tab>		**cntrl, space, blank**
<newline>		**cntrl, space**
<vertical-tab>		**cntrl, space**
<form-feed>		**cntrl, space**
<carriage-return>		**cntrl, space**
<SO>		**cntrl**
<SI>		**cntrl**
<DLE>		**cntrl**
<DC1>		**cntrl**
<DC2>		**cntrl**
<DC3>		**cntrl**
<DC4>		**cntrl**
<NAK>		**cntrl**
<SYN>		**cntrl**
<ETB>		**cntrl**
<CAN>		**cntrl**
<EM>		**cntrl**
<SUB>		**cntrl**
<ESC>		**cntrl**
<IS4>		**cntrl**
<IS3>		**cntrl**
<IS2>		**cntrl**
<IS1>		**cntrl**
<space>		**space, print, blank**
<exclamation-mark>		**punct, print, graph**
<quotation-mark>		**punct, print, graph**
<number-sign>		**punct, print, graph**
<dollar-sign>		**punct, print, graph**
<percent-sign>		**punct, print, graph**
<ampersand>		**punct, print, graph**
<apostrophe>		**punct, print, graph**
<left-parenthesis>		**punct, print, graph**
<right-parenthesis>		**punct, print, graph**
<asterisk>		**punct, print, graph**
<plus-sign>		**punct, print, graph**
<comma>		**punct, print, graph**
<hyphen>		**punct, print, graph**
<period>		**punct, print, graph**

Symbolic Name	Other Case	Character Classes
<slash>		**punct, print, graph**
<zero>		**digit, xdigit, print, graph**
<one>		**digit, xdigit, print, graph**
<two>		**digit, xdigit, print, graph**
<three>		**digit, xdigit, print, graph**
<four>		**digit, xdigit, print, graph**
<five>		**digit, xdigit, print, graph**
<six>		**digit, xdigit, print, graph**
<seven>		**digit, xdigit, print, graph**
<eight>		**digit, xdigit, print, graph**
<nine>		**digit, xdigit, print, graph**
<colon>		**punct, print, graph**
<semicolon>		**punct, print, graph**
<less-than-sign>		**punct, print, graph**
<equals-sign>		**punct, print, graph**
<greater-than-sign>		**punct, print, graph**
<question-mark>		**punct, print, graph**
<commercial-at>		**punct, print, graph**
<A>	<a>	**upper, xdigit, alpha, print, graph**
<B>	<b>	**upper, xdigit, alpha, print, graph**
<C>	<c>	**upper, xdigit, alpha, print, graph**
<D>	<d>	**upper, xdigit, alpha, print, graph**
<E>	<e>	**upper, xdigit, alpha, print, graph**
<F>	<f>	**upper, xdigit, alpha, print, graph**
<G>	<g>	**upper, alpha, print, graph**
<H>	<h>	**upper, alpha, print, graph**
<I>	<i>	**upper, alpha, print, graph**
<J>	<j>	**upper, alpha, print, graph**
<K>	<k>	**upper, alpha, print, graph**
<L>	<l>	**upper, alpha, print, graph**
<M>	<m>	**upper, alpha, print, graph**
<N>	<n>	**upper, alpha, print, graph**
<O>	<o>	**upper, alpha, print, graph**
<P>	<p>	**upper, alpha, print, graph**
<Q>	<q>	**upper, alpha, print, graph**
<R>	<r>	**upper, alpha, print, graph**
<S>	<s>	**upper, alpha, print, graph**
<T>	<t>	**upper, alpha, print, graph**
<U>	<u>	**upper, alpha, print, graph**
<V>	<v>	**upper, alpha, print, graph**
<W>	<w>	**upper, alpha, print, graph**
<X>	<x>	**upper, alpha, print, graph**
<Y>	<y>	**upper, alpha, print, graph**
<Z>	<z>	**upper, alpha, print, graph**
<left-square-bracket>		**punct, print, graph**
<backslash>		**punct, print, graph**
<right-square-bracket>		**punct, print, graph**

Symbolic Name	Other Case	Character Classes
<circumflex>		**punct, print, graph**
<underscore>		**punct, print, graph**
<grave-accent>		**punct, print, graph**
<a>	<A>	**lower, xdigit, alpha, print, graph**
<b>	<B>	**lower, xdigit, alpha, print, graph**
<c>	<C>	**lower, xdigit, alpha, print, graph**
<d>	<D>	**lower, xdigit, alpha, print, graph**
<e>	<E>	**lower, xdigit, alpha, print, graph**
<f>	<F>	**lower, xdigit, alpha, print, graph**
<g>	<G>	**lower, alpha, print, graph**
<h>	<H>	**lower, alpha, print, graph**
<i>	<I>	**lower, alpha, print, graph**
<j>	<J>	**lower, alpha, print, graph**
<k>	<K>	**lower, alpha, print, graph**
<l>	<L>	**lower, alpha, print, graph**
<m>	<M>	**lower, alpha, print, graph**
<n>	<N>	**lower, alpha, print, graph**
<o>	<O>	**lower, alpha, print, graph**
<p>	<P>	**lower, alpha, print, graph**
<q>	<Q>	**lower, alpha, print, graph**
<r>	<R>	**lower, alpha, print, graph**
<s>	<S>	**lower, alpha, print, graph**
<t>	<T>	**lower, alpha, print, graph**
<u>	<U>	**lower, alpha, print, graph**
<v>	<V>	**lower, alpha, print, graph**
<w>	<W>	**lower, alpha, print, graph**
<x>	<X>	**lower, alpha, print, graph**
<y>	<Y>	**lower, alpha, print, graph**
<z>	<Z>	**lower, alpha, print, graph**
<left-curly-bracket>		**punct, print, graph**
<vertical-line>		**punct, print, graph**
<right-curly-bracket>		**punct, print, graph**
<tilde>		**punct, print, graph**
<DEL>		**cntrl**

7.3.2 LC_COLLATE

The *LC_COLLATE* category provides a collation sequence definition for numerous utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x (*sort*, *uniq*, and so on), regular expression matching (see Chapter 9 (on page 165)) and the *strcoll*( ), *strxfrm*( ), *wcscoll*( ), and *wcsxfrm*( ) functions in the System Interfaces volume of IEEE Std 1003.1-200x.

A collation sequence definition shall define the relative order between collating elements (characters and multi-character collating elements) in the locale. This order is expressed in terms of collation values; that is, by assigning each element one or more collation values (also known as collation weights). This does not imply that implementations shall assign such values, but that ordering of strings using the resultant collation definition in the locale behaves as if such assignment is done and used in the collation process. At least the following capabilities are provided:

1. **Multi-character collating elements**. Specification of multi-character collating elements (that is, sequences of two or more characters to be collated as an entity).

2. **User-defined ordering of collating elements**. Each collating element shall be assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching and, unless collation weights are explicitly specified, also as the collation weight to be used in sorting.

3. **Multiple weights and equivalence classes**. Collating elements can be assigned one or more (up to the limit {COLL_WEIGHTS_MAX}, as defined in **<limits.h>**) collating weights for use in sorting. The first weight is hereafter referred to as the primary weight.

4. **One-to-many mapping**. A single character is mapped into a string of collating elements.

5. **Equivalence class definition**. Two or more collating elements have the same collation value (primary weight).

6. **Ordering by weights**. When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements; the elements in each successive pair of elements are then compared according to the relative primary weights for the elements. If equal, and more than one weight has been assigned, then the pairs of collating elements are recompared according to the relative subsequent weights, until either a pair of collating elements compare unequal or the weights are exhausted.

The following keywords shall be recognized in a collation sequence definition. They are described in detail in the following sections.

copy	Specify the name of an existing locale which shall be used as the definition of this category. If this keyword is specified, no other keyword shall be specified.
collating-element	Define a collating-element symbol representing a multi-character collating element. This keyword is optional.
collating-symbol	Define a collating symbol for use in collation order statements. This keyword is optional.
order_start	Define collation rules. This statement shall be followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.
order_end	Specify the end of the collation-order statements.

7.3.2.1 *The collating-element Keyword*

In addition to the collating elements in the character set, the **collating-element** keyword can be used to define multi-character collating elements. The syntax is as follows:

```
"collating-element %s from \"%s\"\n", <collating-symbol>, <string>
```

The <*collating-symbol*> operand shall be a symbolic name, enclosed between angle brackets (’<’ and ’>’), and shall not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand is a string of two or more characters that collates as an entity. A <*collating-element*> defined via this keyword is only recognized with the *LC_COLLATE* category.

For example:

```
collating-element <ch> from "<c><h>"
collating-element <e-acute> from "<acute><e>"
collating-element <ll> from "ll"
```

7.3.2.2 *The collating-symbol Keyword*

This keyword shall be used to define symbols for use in collation sequence statements; that is, between the **order_start** and the **order_end** keywords. The syntax is as follows:

```
"collating-symbol %s\n", <collating-symbol>
```

The <*collating-symbol*> shall be a symbolic name, enclosed between angle brackets (`'<'` and `'>'`), and shall not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. A <*collating-symbol*> defined via this keyword is only recognized within the *LC_COLLATE* category.

For example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

The **collating-symbol** keyword defines a symbolic name that can be associated with a relative position in the character order sequence. While such a symbolic name does not represent any collating element, it can be used as a weight.

7.3.2.3 *The order_start Keyword*

The **order_start** keyword shall precede collation order entries and also define the number of weights for this collation sequence definition and other collation rules. The syntax is as follows:

```
"order_start %s;%s;...;%s\n", <sort-rules>, <sort-rules> ...
```

The operands to the **order_start** keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one **forward** operand is assumed. If present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands shall be separated by semicolons (`';'`). Each operand shall consist of one or more collation directives, separated by commas (`','`). If the number of operands exceeds the {COLL_WEIGHTS_MAX} limit, the utility shall issue a warning message. The following directives shall be supported:

forward Specifies that comparison operations for the weight level shall proceed from start of string towards the end of string.

backward Specifies that comparison operations for the weight level shall proceed from end of string towards the beginning of string.

position Specifies that comparison operations for the weight level shall consider the relative position of elements in the strings not subject to **IGNORE**. The string containing an element not subject to **IGNORE** after the fewest collating elements subject to **IGNORE** from the start of the compare shall collate first. If both strings contain a character not subject to **IGNORE** in the same relative position, the collating values assigned to the elements shall determine the ordering. In case of equality, subsequent characters not subject to **IGNORE** shall be considered in the same manner.

The directives **forward** and **backward** are mutually-exclusive.

If no operands are specified, a single **forward** operand shall be assumed.

For example:

```
order_start    forward;backward
```

7.3.2.4 Collation Order

The **order_start** keyword shall be followed by collating identifier entries. The syntax for the collating element entries is as follows:

```
"%s %s;%s;...;%s\n", <collating-identifier>, <weight>, <weight>, ...
```

Each *collating-identifier* shall consist of either a character (in any of the forms defined in Section 7.3 (on page 120)), a <*collating-element*>, a <*collating-symbol*>, an ellipsis, or the special symbol **UNDEFINED**. The order in which collating elements are specified determines the character order sequence, such that each collating element shall compare less than the elements following it.

A <*collating-element*> shall be used to specify multi-character collating elements, and indicates that the character sequence specified via the <*collating-element*> is to be collated as a unit and in the relative order specified by its place.

A <*collating-symbol*> can be used to define a position in the relative order for use in weights. No weights shall be specified with a <*collating-symbol*>.

The ellipsis symbol specifies that a sequence of characters shall collate according to their encoded character values. It shall be interpreted as indicating that all characters with a coded character set value higher than the value of the character in the preceding line, and lower than the coded character set value for the character in the following line, in the current coded character set, shall be placed in the character collation order between the previous and the following character in ascending order according to their coded character set values. An initial ellipsis shall be interpreted as if the preceding line specified the NUL character, and a trailing ellipsis as if the following line specified the highest coded character set value in the current coded character set. An ellipsis shall be treated as invalid if the preceding or following lines do not specify characters in the current coded character set. The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable between implementations.

The symbol **UNDEFINED** shall be interpreted as including all coded character set values not specified explicitly or via the ellipsis symbol. Such characters shall be inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their coded character set values. If no **UNDEFINED** symbol is specified, and the current coded character set contains characters not specified in this section, the utility shall issue a warning message and place such characters at the end of the character collation order.

The optional operands for each collation-element shall be used to define the primary, secondary, or subsequent weights for the collating element. The first operand specifies the relative primary weight, the second the relative secondary weight, and so on. Two or more collation-elements can be assigned the same weight; they belong to the same *equivalence class* if they have the same primary weight. Collation shall behave as if, for each weight level, elements subject to **IGNORE** are removed, unless the **position** collation directive is specified for the corresponding level with the **order_start** keyword. Then each successive pair of elements shall be compared according to the relative weights for the elements. If the two strings compare equal, the process shall be repeated for the next weight level, up to the limit {COLL_WEIGHTS_MAX}.

Weights shall be expressed as characters (in any of the forms specified in Section 7.3 (on page 120)), <*collating-symbol*>s, <*collating-element*>s, an ellipsis, or the special symbol **IGNORE**. A single character, a <*collating-symbol*>, or a <*collating-element*> shall represent the relative position in the character collating sequence of the character or symbol, rather than the character or characters themselves. Thus, rather than assigning absolute values to weights, a particular weight is expressed using the relative order value assigned to a collating element based on its order in the character collation sequence.

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. For example, if the <eszet> is given the string `"<s><s>"` as a weight, comparisons are performed as if all occurrences of the <eszet> are replaced by `"<s><s>"` (assuming that `"<s>"` has the collating weight `"<s>"`). If it is necessary to define <eszet> and `"<s><s>"` as an equivalence class, then a collating element must be defined for the string `"ss"`.

All characters specified via an ellipsis shall by default be assigned unique weights, equal to the relative order of characters. Characters specified via an explicit or implicit **UNDEFINED** special symbol shall by default be assigned the same primary weight (that is, they belong to the same equivalence class). An ellipsis symbol as a weight shall be interpreted to mean that each character in the sequence shall have unique weights, equal to the relative order of their character in the character collation sequence. The use of the ellipsis as a weight shall be treated as an error if the collating element is neither an ellipsis nor the special symbol **UNDEFINED**.

The special keyword **IGNORE** as a weight shall indicate that when strings are compared using the weights at the level where **IGNORE** is specified, the collating element shall be ignored; that is, as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are subject to **IGNORE** in their primary weight form an equivalence class.

An empty operand shall be interpreted as the collating element itself.

For example, the order statement:

```
<a>     <a>;<a>
```

is equal to:

```
<a>
```

An ellipsis can be used as an operand if the collating element was an ellipsis, and shall be interpreted as the value of each character defined by the ellipsis.

The collation order as defined in this section affects the interpretation of bracket expressions in regular expressions (see Section 9.3.5 (on page 168)).

For example:

```
order_start  forward;backward
UNDEFINED    IGNORE;IGNORE
<LOW>
<space>      <LOW>;<space>
...          <LOW>;...
<a>          <a>;<a>
<a-acute>    <a>;<a-acute>
<a-grave>    <a>;<a-grave>
<A>          <a>;<A>
<A-acute>    <a>;<A-acute>
<A-grave>    <a>;<A-grave>
<ch>         <ch>;<ch>
<Ch>         <ch>;<Ch>
<s>          <s>;<s>
<eszet>      "<s><s>";"<eszet><eszet>"
order_end
```

This example is interpreted as follows:

1. The **UNDEFINED** means that all characters not specified in this definition (explicitly or via the ellipsis) shall be ignored for collation purposes.
2. All characters between <space> and `'a'` shall have the same primary equivalence class and individual secondary weights based on their ordinal encoded values.
3. All characters based on the uppercase or lowercase character `'a'` belong to the same primary equivalence class.
4. The multi-character collating element <ch> is represented by the collating symbol <ch> and belongs to the same primary equivalence class as the multi-character collating element <Ch>.

7.3.2.5 The order_end Keyword

The collating order entries shall be terminated with an **order_end** keyword.

7.3.2.6 LC_COLLATE Category in the POSIX Locale

The collation sequence definition of the POSIX locale follows; the code listing depicts the *localedef* input.

```
LC_COLLATE
# This is the POSIX locale definition for the LC_COLLATE category.
# The order is the same as in the ASCII codeset.
order_start forward
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
<CAN>
<EM>
<SUB>
<ESC>
```

```
    <IS4>
    <IS3>
    <IS2>
    <IS1>
    <space>
    <exclamation-mark>
    <quotation-mark>
    <number-sign>
    <dollar-sign>
    <percent-sign>
    <ampersand>
    <apostrophe>
    <left-parenthesis>
    <right-parenthesis>
    <asterisk>
    <plus-sign>
    <comma>
    <hyphen>
    <period>
    <slash>
    <zero>
    <one>
    <two>
    <three>
    <four>
    <five>
    <six>
    <seven>
    <eight>
    <nine>
    <colon>
    <semicolon>
    <less-than-sign>
    <equals-sign>
    <greater-than-sign>
    <question-mark>
    <commercial-at>
    <A>
    <B>
    <C>
    <D>
    <E>
    <F>
    <G>
    <H>
    <I>
    <J>
    <K>
    <L>
    <M>
    <N>
    <O>
```

```
<P>
<Q>
<R>
<S>
<T>
<U>
<V>
<W>
<X>
<Y>
<Z>
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underscore>
<grave-accent>
<a>
<b>
<c>
<d>
<e>
<f>
<g>
<h>
<i>
<j>
<k>
<l>
<m>
<n>
<o>
<p>
<q>
<r>
<s>
<t>
<u>
<v>
<w>
<x>
<y>
<z>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
order_end
#
END LC_COLLATE
```

7.3.3 LC_MONETARY

The *LC_MONETARY* category shall define the rules and symbols that are used to format monetary numeric information. This information is available through the *localeconv*( ) function XSI and is used by the *strfmon*( ) function.

XSI Some of the information is also available in an alternative form via the *nl_langinfo*( ) function (see CRNCYSTR in **<langinfo.h>**).

The following items are defined in this category of the locale. The item names are the keywords recognized by the *localedef* utility when defining a locale. They are also similar to the member names of the **lconv** structure defined in **<locale.h>**; see **<locale.h>** for the exact symbols in the header. The *localeconv*( ) function returns {CHAR_MAX} for unspecified integer items and the empty string ("") for unspecified or size zero string items.

In a locale definition file, the operands are strings, formatted as indicated by the grammar in Section 7.4 (on page 149). For some keywords, the strings can contain only integers. Keywords that are not provided, string values set to the empty string (""), or integer keywords set to −1, are used to indicate that the value is not available in the locale. The following keywords shall be recognized:

copy	Specify the name of an existing locale which shall be used as the definition of this category. If this keyword is specified, no other keyword shall be specified. **Note:** This is a *localedef* utility keyword, unavailable through *localeconv*( ).
int_curr_symbol	The international currency symbol. The operand shall be a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in the ISO 4217:1995 standard. The fourth character shall be the character used to separate the international currency symbol from the monetary quantity.
currency_symbol	The string that shall be used as the local currency symbol.
mon_decimal_point	The operand is a string containing the symbol that shall be used as the decimal delimiter (radix character) in monetary formatted quantities.
mon_thousands_sep	The operand is a string containing the symbol that shall be used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities.
mon_grouping	Define the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not −1, then the size of the previous group (if any) shall be repeatedly used for the remainder of the digits. If the last integer is −1, then no further grouping shall be performed.
positive_sign	A string that shall be used to indicate a non-negative-valued formatted monetary quantity.
negative_sign	A string that shall be used to indicate a negative-valued formatted monetary quantity.
int_frac_digits	An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity

using **int_curr_symbol**.

frac_digits	An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using **currency_symbol**.
p_cs_precedes	An integer set to 1 if the **currency_symbol** precedes the value for a monetary quantity with a non-negative value, and set to 0 if the symbol succeeds the value.
p_sep_by_space	An integer set to 0 if no space separates the **currency_symbol** from the value for a monetary quantity with a non-negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.
n_cs_precedes	An integer set to 1 if the **currency_symbol** precedes the value for a monetary quantity with a negative value, and set to 0 if the symbol succeeds the value.
n_sep_by_space	An integer set to 0 if no space separates the **currency_symbol** from the value for a monetary quantity with a negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.
p_sign_posn	An integer set to a value indicating the positioning of the **positive_sign** for a monetary quantity with a non-negative value. The following integer values shall be recognized for **int_n_sign_posn**, **int_p_sign_posn**, **n_sign_posn**, and **p_sign_posn**: 0 Parentheses enclose the quantity and the **currency_symbol**. 1 The sign string precedes the quantity and the **currency_symbol**. 2 The sign string succeeds the quantity and the **currency_symbol**. 3 The sign string precedes the **currency_symbol**. 4 The sign string succeeds the **currency_symbol**.
n_sign_posn	An integer set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.
int_p_cs_precedes	An integer set to 1 if the **int_curr_symbol** precedes the value for a monetary quantity with a non-negative value, and set to 0 if the symbol succeeds the value.
int_n_cs_precedes	An integer set to 1 if the **int_curr_symbol** precedes the value for a monetary quantity with a negative value, and set to 0 if the symbol succeeds the value.
int_p_sep_by_space	An integer to set 0 if no space separates the **int_curr_symbol** from the value for a monetary quantity with a non-negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.
int_n_sep_by_space	An integer set to 0 if no space separates the **int_curr_symbol** from the value for a monetary quantity with a negative value, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

int_p_sign_posn	An integer set to a value indicating the positioning of the **positive_sign** for a positive monetary quantity formatted with the international format.
int_n_sign_posn	An integer set to a value indicating the positioning of the **negative_sign** for a negative monetary quantity formatted with the international format.

7.3.3.1 *LC_MONETARY Category in the POSIX Locale*

The monetary formatting definitions for the POSIX locale follow; the code listing depicting the *localedef* input, the table representing the same information with the addition of *localeconv*( ) XSI and *nl_langinfo*( ) formats. All values are unspecified in the POSIX locale.

```
LC_MONETARY
# This is the POSIX locale definition for
# the LC_MONETARY category.
#
int_curr_symbol     ""
currency_symbol     ""
mon_decimal_point   ""
mon_thousands_sep   ""
mon_grouping        -1
positive_sign       ""
negative_sign       ""
int_frac_digits     -1
frac_digits         -1
p_cs_precedes       -1
p_sep_by_space      -1
n_cs_precedes       -1
n_sep_by_space      -1
p_sign_posn         -1
n_sign_posn         -1
#
END LC_MONETARY
```

Item	langinfo Constant	POSIX Locale Value	localeconv( ) Value	localedef Value
int_curr_symbol	—	N/A	""	""
currency_symbol	CRNCYSTR	N/A	""	""
mon_decimal_point	—	N/A	""	""
mon_thousands_sep	—	N/A	""	""
mon_grouping	—	N/A	""	""
positive_sign	—	N/A	""	""
negative_sign	—	N/A	""	""
int_frac_digits	—	N/A	{CHAR_MAX}	–1
frac_digits	—	N/A	{CHAR_MAX}	–1
p_cs_precedes	CRNCYSTR	N/A	{CHAR_MAX}	–1
p_sep_by_space	—	N/A	{CHAR_MAX}	–1
n_cs_precedes	CRNCYSTR	N/A	{CHAR_MAX}	–1
n_sep_by_space	—	N/A	{CHAR_MAX}	–1
p_sign_posn	—	N/A	{CHAR_MAX}	–1
n_sign_posn	—	N/A	{CHAR_MAX}	–1

XSI In the preceding table, the **langinfo Constant** column represents an XSI-conformant extension. The entry N/A indicates that the value is not available in the POSIX locale.

7.3.4 LC_NUMERIC

The *LC_NUMERIC* category shall define the rules and symbols that are used to format non-monetary numeric information. This information is available through the *localeconv*( ) function. XSI Some of the information is also available in an alternative form via the *nl_langinfo*( ) function.

The following items are defined in this category of the locale. The item names are the keywords recognized by the *localedef* utility when defining a locale. They are also similar to the member names of the **lconv** structure defined in **<locale.h>**; see **<locale.h>** for the exact symbols in the header. The *localeconv*( ) function returns {CHAR_MAX} for unspecified integer items and the empty string ("") for unspecified or size zero string items.

In a locale definition file, the operands are strings, formatted as indicated by the grammar in Section 7.4 (on page 149). For some keywords, the strings can only contain integers. Keywords that are not provided, string values set to the empty string (""), or integer keywords set to −1, shall be used to indicate that the value is not available in the locale. The following keywords shall be recognized:

copy Specify the name of an existing locale which shall be used as the definition of this category. If this keyword is specified, no other keyword shall be specified. |

Note: This is a *localedef* utility keyword, unavailable through *localeconv*( ).

decimal_point The operand is a string containing the symbol that shall be used as the decimal delimiter (radix character) in numeric, non-monetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string. In contexts where standards limit the **decimal_point** to a single byte, the result of specifying a multi-byte operand shall be unspecified.

thousands_sep The operand is a string containing the symbol that shall be used as a separator for groups of digits to the left of the decimal delimiter in numeric, non-monetary formatted monetary quantities. In contexts where standards limit the **thousands_sep** to a single byte, the result of specifying a multi-byte operand shall be unspecified.

grouping Define the size of each group of digits in formatted non-monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not −1, then the size of the previous group (if any) shall be repeatedly used for the remainder of the digits. If the last integer is −1, then no further grouping shall be performed.

7.3.4.1 LC_NUMERIC Category in the POSIX Locale

The non-monetary numeric formatting definitions for the POSIX locale follow; the code listing depicting the *localedef* input, the table representing the same information with the addition of XSI *localeconv*( ) values, and *nl_langinfo*( ) constants.

```
LC_NUMERIC
# This is the POSIX locale definition for
# the LC_NUMERIC category.
#
decimal_point    "<period>"
thousands_sep    ""
grouping         -1
#
```

```
END LC_NUMERIC
```

Item	langinfo Constant	POSIX Locale Value	localeconv( ) Value	localedef Value
decimal_point	RADIXCHAR	"."	"."	.
thousands_sep	THOUSEP	N/A	""	""
grouping	—	N/A	""	−1

XSI In the preceding table, the **langinfo Constant** column represents an XSI-conforming extension. The entry N/A indicates that the value is not available in the POSIX locale.

7.3.5 LC_TIME

XSI The *LC_TIME* category shall define the interpretation of the conversion specifications supported by the *date* utility and shall affect the behavior of the *strftime*( ), *wcsftime*( ), *strptime*( ), and *nl_langinfo*( ) functions. Since the interfaces for C-language access and locale definition differ significantly, they are described separately.

7.3.5.1 LC_TIME Locale Definition

For locale definition, the following mandatory keywords shall be recognized:

copy	Specify the name of an existing locale which shall be used as the definition of this category. If this keyword is specified, no other keyword shall be specified.
abday	Define the abbreviated weekday names, corresponding to the `%a` conversion specification (conversion specification in the *strftime*( ), *wcsftime*( ), and *strptime*( ) functions). The operand shall consist of seven semicolon-separated strings, each surrounded by double-quotes. The first string shall be the abbreviated name of the day corresponding to Sunday, the second the abbreviated name of the day corresponding to Monday, and so on.
day	Define the full weekday names, corresponding to the `%A` conversion specification. The operand shall consist of seven semicolon-separated strings, each surrounded by double-quotes. The first string is the full name of the day corresponding to Sunday, the second the full name of the day corresponding to Monday, and so on.
abmon	Define the abbreviated month names, corresponding to the `%b` conversion specification. The operand shall consist of twelve semicolon-separated strings, each surrounded by double-quotes. The first string shall be the abbreviated name of the first month of the year (January), the second the abbreviated name of the second month, and so on.
mon	Define the full month names, corresponding to the `%B` conversion specification. The operand shall consist of twelve semicolon-separated strings, each surrounded by double-quotes. The first string shall be the full name of the first month of the year (January), the second the full name of the second month, and so on.
d_t_fmt	Define the appropriate date and time representation, corresponding to the `%c` conversion specification. The operand shall consist of a string containing any combination of characters and conversion specifications. In addition, the string can contain escape sequences defined in the table in Table 5-1 (on page 108) (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`).

d_fmt Define the appropriate date representation, corresponding to the `%x` conversion specification. The operand shall consist of a string containing any combination of characters and conversion specifications. In addition, the string can contain escape sequences defined in the table in Table 5-1 (on page 108).

t_fmt Define the appropriate time representation, corresponding to the `%X` conversion specification. The operand shall consist of a string containing any combination of characters and conversion specifications. In addition, the string can contain escape sequences defined in the table in Table 5-1 (on page 108).

am_pm Define the appropriate representation of the *ante meridiem* and *post meridiem* strings, corresponding to the `%p` conversion specification. The operand shall consist of two strings, separated by a semicolon, each surrounded by double-quotes. The first string shall represent the *ante meridiem* designation, the last string the *post meridiem* designation.

t_fmt_ampm Define the appropriate time representation in the 12-hour clock format with **am_pm**, corresponding to the `%r` conversion specification. The operand shall consist of a string and can contain any combination of characters and conversion specifications. If the string is empty, the 12-hour format is not supported in the locale.

era Define how years are counted and displayed for each era in a locale. The operand shall consist of semicolon-separated strings. Each string shall be an era description segment with the format:

`direction:offset:start_date:end_date:era_name:era_format`

according to the definitions below. There can be as many era description segments as are necessary to describe the different eras.

Note: The start of an era might not be the earliest point in the era—it may be the latest. For example, the Christian era BC starts on the day before January 1, AD 1, and increases with earlier time.

direction Either a `'+'` or a `'−'` character. The `'+'` character shall indicate that years closer to the *start_date* have lower numbers than those closer to the *end_date*. The `'−'` character shall indicate that years closer to the *start_date* have higher numbers than those closer to the *end_date*.

offset The number of the year closest to the *start_date* in the era, corresponding to the `%Ey` conversion specification.

start_date A date in the form *yyyy*/*mm*/*dd*, where *yyyy*, *mm*, and *dd* are the year, month, and day numbers respectively of the start of the era. Years prior to AD 1 shall be represented as negative numbers.

end_date The ending date of the era, in the same format as the *start_date*, or one of the two special values `"−*"` or `"+*"`. The value `"−*"` shall indicate that the ending date is the beginning of time. The value `"+*"` shall indicate that the ending date is the end of time.

era_name A string representing the name of the era, corresponding to the `%EC` conversion specification.

era_format	A string for formatting the year in the era, corresponding to the `%EY` conversion specification.
era_d_fmt	Define the format of the date in alternative era notation, corresponding to the `%Ex` conversion specification.
era_t_fmt	Define the locale's appropriate alternative time format, corresponding to the `%EX` conversion specification.
era_d_t_fmt	Define the locale's appropriate alternative date and time format, corresponding to the `%Ec` conversion specification.
alt_digits	Define alternative symbols for digits, corresponding to the `%O` modified conversion specification. The operand shall consist of semicolon-separated strings, each surrounded by double-quotes. The first string shall be the alternative symbol corresponding with zero, the second string the symbol corresponding with one, and so on. Up to 100 alternative symbol strings can be specified. The `%O` modifier shall indicate that the string corresponding to the value specified via the conversion specification shall be used instead of the value.

7.3.5.2 LC_TIME C-Language Access

XSI This section describes extensions to access information in the *LC_TIME* category using the *nl_langinfo*( ) function. This functionality is dependent on support of the XSI extension (and the rest of this section is not further shaded for this option).

The following constants used to identify items of *langinfo* data can be used as arguments to the *nl_langinfo*( ) function to access information in the *LC_TIME* category. These constants are defined in the **<langinfo.h>** header.

ABDAY_*x*	The abbreviated weekday names (for example Sun), where *x* is a number from 1 to 7.
DAY_*x*	The full weekday names (for example Sunday), where *x* is a number from 1 to 7.
ABMON_*x*	The abbreviated month names (for example Jan), where *x* is a number from 1 to 12.
MON_*x*	The full month names (for example January), where *x* is a number from 1 to 12.
D_T_FMT	The appropriate date and time representation.
D_FMT	The appropriate date representation.
T_FMT	The appropriate time representation.
AM_STR	The appropriate ante-meridiem affix.
PM_STR	The appropriate post-meridiem affix.
T_FMT_AMPM	The appropriate time representation in the 12-hour clock format with AM_STR and PM_STR.
ERA	The era description segments, which describe how years are counted and displayed for each era in a locale. Each era description segment shall have the format:

`direction:offset:start_date:end_date:era_name:era_format`

according to the definitions below. There can be as many era description segments as are necessary to describe the different eras. Era description segments are separated by semicolons.

direction	Either a `'+'` or a `'−'` character. The `'+'` character shall indicate that years closer to the *start_date* have lower numbers than those closer to the *end_date*. The `'−'` character shall indicate that years closer to the *start_date* have higher numbers than those closer to the *end_date*.
offset	The number of the year closest to the *start_date* in the era.
start_date	A date in the form *yyyy*/*mm*/*dd*, where *yyyy*, *mm*, and *dd* are the year, month, and day numbers respectively of the start of the era. Years prior to AD 1 shall be represented as negative numbers.
end_date	The ending date of the era, in the same format as the *start_date*, or one of the two special values `"−*"` or `"+*"`. The value `"−*"` shall indicate that the ending date is the beginning of time. The value `"+*"` shall indicate that the ending date is the end of time.
era_name	The era, corresponding to the `%EC` conversion specification.
era_format	The format of the year in the era, corresponding to the `%EY` conversion specification.

ERA_D_FMT	The era date format.
ERA_T_FMT	The locale's appropriate alternative time format, corresponding to the `%EX` conversion specification.
ERA_D_T_FMT	The locale's appropriate alternative date and time format, corresponding to the `%Ec` conversion specification.
ALT_DIGITS	The alternative symbols for digits, corresponding to the `%O` conversion specification modifier. The value consists of semicolon-separated symbols. The first is the alternative symbol corresponding to zero, the second is the symbol corresponding to one, and so on. Up to 100 alternative symbols may be specified.

7.3.5.3 *LC_TIME Category in the POSIX Locale*

The *LC_TIME* category definition of the POSIX locale follows; the code listing depicts the *localedef* input; the table represents the same information with the addition of *localedef* keywords, conversion specifiers used by the *date* utility and the *strftime*( ), *wcsftime*( ), and *strptime*( ) functions, XSI and *nl_langinfo*( ) constants.

```
LC_TIME
# This is the POSIX locale definition for
# the LC_TIME category.
#
# Abbreviated weekday names (%a)
abday      "<S><u><n>";"<M><o><n>";"<T><u><e>";"<W><e><d>";\
           "<T><h><u>";"<F><r><i>";"<S><a><t>"
#
# Full weekday names (%A)
```

```
day        "<S><u><n><d><a><y>";"<M><o><n><d><a><y>";\
           "<T><u><e><s><d><a><y>";"<W><e><d><n><e><s><d><a><y>";\
           "<T><h><u><r><s><d><a><y>";"<F><r><i><d><a><y>";\
           "<S><a><t><u><r><d><a><y>"
#
# Abbreviated month names (%b)
abmon      "<J><a><n>";"<F><e><b>";"<M><a><r>";\
           "<A><p><r>";"<M><a><y>";"<J><u><n>";\
           "<J><u><l>";"<A><u><g>";"<S><e><p>";\
           "<O><c><t>";"<N><o><v>";"<D><e><c>"
#
# Full month names (%B)
mon        "<J><a><n><u><a><r><y>";"<F><e><b><r><u><a><r><y>";\
           "<M><a><r><c><h>";"<A><p><r><i><l>";\
           "<M><a><y>";"<J><u><n><e>";\
           "<J><u><l><y>";"<A><u><g><u><s><t>";\
           "<S><e><p><t><e><m><b><e><r>";"<O><c><t><o><b><e><r>";\
           "<N><o><v><e><m><b><e><r>";"<D><e><c><e><m><b><e><r>"
#
# Equivalent of AM/PM (%p)      "AM";"PM"
am_pm      "<A><M>";"<P><M>"
#
# Appropriate date and time representation (%c)
#    "%a %b %e %H:%M:%S %Y"
d_t_fmt    "<percent-sign><a><space><percent-sign><b>\
            <space><percent-sign><e><space><percent-sign><H>\
            <colon><percent-sign><M><colon><percent-sign><S>\
            <space><percent-sign><Y>"
#
# Appropriate date representation (%x)   "%m/%d/%y"
d_fmt      "<percent-sign><m><slash><percent-sign><d>\
            <slash><percent-sign><y>"
#
# Appropriate time representation (%X)   "%H:%M:%S"
t_fmt      "<percent-sign><H><colon><percent-sign><M>\
            <colon><percent-sign><S>"
#
# Appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon>\
            <percent-sign><S><space><percent_sign><p>"
#
END LC_TIME
```

localedef Keyword	langinfo Constant	Conversion Specification	POSIX Locale Value
d_t_fmt	D_T_FMT	`%c`	`"%a %b %e %H:%M:%S %Y"`
d_fmt	D_FMT	`%x`	`"%m/%d/%y"`
t_fmt	T_FMT	`%X`	`"%H:%M:%S"`

localedef Keyword	**langinfo Constant**	**Conversion Specification**	**POSIX Locale Value**
am_pm	AM_STR	%p	"AM"
am_pm	PM_STR	%p	"PM"
t_fmt_ampm	T_FMT_AMPM	%r	"%I:%M:%S %p"
day	DAY_1	%A	"Sunday"
day	DAY_2	%A	"Monday"
day	DAY_3	%A	"Tuesday"
day	DAY_4	%A	"Wednesday"
day	DAY_5	%A	"Thursday"
day	DAY_6	%A	"Friday"
day	DAY_7	%A	"Saturday"
abday	ABDAY_1	%a	"Sun"
abday	ABDAY_2	%a	"Mon"
abday	ABDAY_3	%a	"Tue"
abday	ABDAY_4	%a	"Wed"
abday	ABDAY_5	%a	"Thu"
abday	ABDAY_6	%a	"Fri"
abday	ABDAY_7	%a	"Sat"
mon	MON_1	%B	"January"
mon	MON_2	%B	"February"
mon	MON_3	%B	"March"
mon	MON_4	%B	"April"
mon	MON_5	%B	"May"
mon	MON_6	%B	"June"
mon	MON_7	%B	"July"
mon	MON_8	%B	"August"
mon	MON_9	%B	"September"
mon	MON_10	%B	"October"
mon	MON_11	%B	"November"
mon	MON_12	%B	"December"
abmon	ABMON_1	%b	"Jan"
abmon	ABMON_2	%b	"Feb"
abmon	ABMON_3	%b	"Mar"
abmon	ABMON_4	%b	"Apr"
abmon	ABMON_5	%b	"May"
abmon	ABMON_6	%b	"Jun"
abmon	ABMON_7	%b	"Jul"
abmon	ABMON_8	%b	"Aug"
abmon	ABMON_9	%b	"Sep"
abmon	ABMON_10	%b	"Oct"
abmon	ABMON_11	%b	"Nov"
abmon	ABMON_12	%b	"Dec"
era	ERA	%EC, %Ey, %EY	N/A
era_d_fmt	ERA_D_FMT	%Ex	N/A
era_t_fmt	ERA_T_FMT	%EX	N/A
era_d_t_fmt	ERA_D_T_FMT	%Ec	N/A
alt_digits	ALT_DIGITS	%O	N/A

XSI In the preceding table, the **langinfo Constant** column represents an XSI-conformant extension.

The entry ''N/A'' indicates the value is not available in the POSIX locale.

7.3.6 LC_MESSAGES

The *LC_MESSAGES* category shall define the format and values used by various utilities for affirmative and negative responses. XSI This information is available through the *nl_langinfo*( ) function.

XSI The message catalog used by the standard utilities and selected by the *catopen*( ) function shall be determined by the setting of *NLSPATH*; see Chapter 8 (on page 157). The *LC_MESSAGES* category can be specified as part of an *NLSPATH* substitution field.

The following keywords shall be recognized as part of the locale definition file.

copy Specify the name of an existing locale which shall be used as the definition of this category. If this keyword is specified, no other keyword shall be specified. |

Note: This is a *localedef* keyword, unavailable through *nl_langinfo*( ).

yesexpr The operand consists of an extended regular expression (see Section 9.4 (on page 171)) that describes the acceptable affirmative response to a question expecting an affirmative or negative response.

noexpr The operand consists of an extended regular expression that describes the acceptable negative response to a question expecting an affirmative or negative response.

7.3.6.1 *LC_MESSAGES Category for the POSIX Locale*

The format and values for affirmative and negative responses of the POSIX locale follow; the code listing depicting the *localedef* input, the table representing the same information XSI with the addition of *nl_langinfo*( ) constants.

```
LC_MESSAGES
# This is the POSIX locale definition for
# the LC_MESSAGES category.
#
yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
#
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"
#
END LC_MESSAGES
```

localedef Keyword	langinfo Constant	POSIX Locale Value
yesexpr	YESEXPR	`"^[yY]"`
noexpr	NOEXPR	`"^[nN]"`

XSI In the preceding table, the **langinfo Constant** column represents an XSI-conformant extension.

7.4 Locale Definition Grammar

The grammar and lexical conventions in this section shall together describe the syntax for the locale definition source. The general conventions for this style of grammar are described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 1.10, Grammar Conventions. The grammar shall take precedence over the text in this chapter.

7.4.1 Locale Lexical Conventions

The lexical conventions for the locale definition grammar are described in this section.

The following tokens shall be processed (in addition to those string constants shown in the grammar):

LOC_NAME	A string of characters representing the name of a locale.
CHAR	Any single character.
NUMBER	A decimal number, represented by one or more decimal digits.
COLLSYMBOL	A symbolic name, enclosed between angle brackets. The string cannot duplicate any charmap symbol defined in the current charmap (if any), or a **COLLELEMENT** symbol.
COLLELEMENT	A symbolic name, enclosed between angle brackets, which cannot duplicate either any charmap symbol or a **COLLSYMBOL** symbol.
CHARCLASS	A string of alphanumeric characters from the portable character set, the first of which is not a digit, consisting of at least one and at most {CHARCLASS_NAME_MAX} bytes, and optionally surrounded by double-quotes.
CHARSYMBOL	A symbolic name, enclosed between angle brackets, from the current charmap (if any).
OCTAL_CHAR	One or more octal representations of the encoding of each byte in a single character. The octal representation consists of an escape character (normally a backslash) followed by two or more octal digits.
HEX_CHAR	One or more hexadecimal representations of the encoding of each byte in a single character. The hexadecimal representation consists of an escape character followed by the constant *x* and two or more hexadecimal digits.
DECIMAL_CHAR	One or more decimal representations of the encoding of each byte in a single character. The decimal representation consists of an escape character followed by a character `'d'` and two or more decimal digits.
ELLIPSIS	The string `"..."`.
EXTENDED_REG_EXP	An extended regular expression as defined in the grammar in Section 9.5 (on page 175).
EOL	The line termination character newline.

7.4.2 Locale Grammar

This section presents the grammar for the locale definition.

```
%token              LOC_NAME
%token              CHAR
%token              NUMBER
%token              COLLSYMBOL COLLELEMENT
%token              CHARSYMBOL OCTAL_CHAR HEX_CHAR DECIMAL_CHAR
%token              ELLIPSIS
%token              EXTENDED_REG_EXP
%token              EOL

%start              locale_definition

%%

locale_definition   : global_statements locale_categories
                    |                   locale_categories
                    ;

global_statements   : global_statements symbol_redefine
                    | symbol_redefine
                    ;

symbol_redefine     : 'escape_char'  CHAR EOL
                    | 'comment_char' CHAR EOL
                    ;

locale_categories   : locale_categories locale_category
                    | locale_category
                    ;

locale_category     : lc_ctype | lc_collate | lc_messages
                    | lc_monetary | lc_numeric | lc_time
                    ;

/* The following grammar rules are common to all categories */

char_list           : char_list char_symbol
                    | char_symbol
                    ;

char_symbol         : CHAR | CHARSYMBOL
                    | OCTAL_CHAR | HEX_CHAR | DECIMAL_CHAR
                    ;

elem_list           : elem_list char_symbol
                    | elem_list COLLSYMBOL
                    | elem_list COLLELEMENT
                    | char_symbol
                    | COLLSYMBOL
                    | COLLELEMENT
                    ;

symb_list           : symb_list COLLSYMBOL
                    | COLLSYMBOL
                    ;
```

```
locale_name         : LOC_NAME
                    | ’"’ LOC_NAME ’"’
                    ;

/* The following is the LC_CTYPE category grammar */

lc_ctype            : ctype_hdr ctype_keywords         ctype_tlr
                    | ctype_hdr ’copy’ locale_name EOL ctype_tlr
                    ;

ctype_hdr           : ’LC_CTYPE’ EOL
                    ;

ctype_keywords      : ctype_keywords ctype_keyword
                    | ctype_keyword
                    ;

ctype_keyword       : charclass_keyword charclass_list EOL
                    | charconv_keyword charconv_list EOL
                    | ’charclass’ charclass_namelist EOL
                    ;

charclass_namelist  : charclass_namelist ’;’ CHARCLASS
                    | CHARCLASS
                    ;

charclass_keyword   : ’upper’ | ’lower’ | ’alpha’ | ’digit’
                    | ’punct’ | ’xdigit’ | ’space’ | ’print’
                    | ’graph’ | ’blank’ | ’cntrl’ | ’alnum’
                    | CHARCLASS
                    ;

charclass_list      : charclass_list ’;’ char_symbol
                    | charclass_list ’;’ ELLIPSIS ’;’ char_symbol
                    | char_symbol
                    ;

charconv_keyword    : ’toupper’
                    | ’tolower’
                    ;

charconv_list       : charconv_list ’;’ charconv_entry
                    | charconv_entry
                    ;

charconv_entry      : ’(’ char_symbol ’,’ char_symbol ’)’
                    ;

ctype_tlr           : ’END’ ’LC_CTYPE’ EOL
                    ;

/* The following is the LC_COLLATE category grammar */

lc_collate          : collate_hdr collate_keywords       collate_tlr
                    | collate_hdr ’copy’ locale_name EOL collate_tlr
                    ;

collate_hdr         : ’LC_COLLATE’ EOL
                    ;
```

```
collate_keywords    :                order_statements
                    | opt_statements order_statements
                    ;

opt_statements      : opt_statements collating_symbols
                    | opt_statements collating_elements
                    | collating_symbols
                    | collating_elements
                    ;

collating_symbols   : ’collating-symbol’ COLLSYMBOL EOL
                    ;

collating_elements  : ’collating-element’ COLLELEMENT
                    | ’from’ ’"’ elem_list ’"’ EOL
                    ;

order_statements    : order_start collation_order order_end
                    ;

order_start         : ’order_start’ EOL
                    | ’order_start’ order_opts EOL
                    ;

order_opts          : order_opts ’;’ order_opt
                    | order_opt
                    ;

order_opt           : order_opt ’,’ opt_word
                    | opt_word
                    ;

opt_word            : ’forward’ | ’backward’ | ’position’
                    ;

collation_order     : collation_order collation_entry
                    | collation_entry
                    ;

collation_entry     : COLLSYMBOL EOL
                    | collation_element weight_list EOL
                    | collation_element             EOL
                    ;

collation_element   : char_symbol
                    | COLLELEMENT
                    | ELLIPSIS
                    | ’UNDEFINED’
                    ;

weight_list         : weight_list ’;’ weight_symbol
                    | weight_list ’;’
                    | weight_symbol
                    ;

weight_symbol       : /* empty */
                    | char_symbol
                    | COLLSYMBOL
                    | ’"’ elem_list ’"’
```

```
                    | ’"’ symb_list ’"’
                    | ELLIPSIS
                    | ’IGNORE’
                    ;

order_end           : ’order_end’ EOL
                    ;

collate_tlr         : ’END’ ’LC_COLLATE’ EOL
                    ;

/* The following is the LC_MESSAGES category grammar */

lc_messages         : messages_hdr messages_keywords      messages_tlr
                    | messages_hdr ’copy’ locale_name EOL messages_tlr
                    ;

messages_hdr        : ’LC_MESSAGES’ EOL
                    ;

messages_keywords   : messages_keywords messages_keyword
                    | messages_keyword
                    ;

messages_keyword    : ’yesexpr’ ’"’ EXTENDED_REG_EXP ’"’ EOL
                    | ’noexpr’  ’"’ EXTENDED_REG_EXP ’"’ EOL
                    ;

messages_tlr        : ’END’ ’LC_MESSAGES’ EOL
                    ;

/* The following is the LC_MONETARY category grammar */

lc_monetary         : monetary_hdr monetary_keywords       monetary_tlr
                    | monetary_hdr ’copy’ locale_name EOL  monetary_tlr
                    ;

monetary_hdr        : ’LC_MONETARY’ EOL
                    ;

monetary_keywords   : monetary_keywords monetary_keyword
                    | monetary_keyword
                    ;

monetary_keyword    : mon_keyword_string mon_string EOL
                    | mon_keyword_char NUMBER EOL
                    | mon_keyword_char ’-1’   EOL
                    | mon_keyword_grouping mon_group_list EOL
                    ;

mon_keyword_string  : ’int_curr_symbol’ | ’currency_symbol’
                    | ’mon_decimal_point’ | ’mon_thousands_sep’
                    | ’positive_sign’ | ’negative_sign’
                    ;

mon_string          : ’"’ char_list ’"’
                    | ’""’
                    ;
```

```
mon_keyword_char    : ’int_frac_digits’ | ’frac_digits’
                    | ’p_cs_precedes’ | ’p_sep_by_space’
                    | ’n_cs_precedes’ | ’n_sep_by_space’
                    | ’p_sign_posn’ | ’n_sign_posn’
                    ;

mon_keyword_grouping : ’mon_grouping’
                    ;

mon_group_list      : NUMBER
                    | mon_group_list ’;’ NUMBER
                    ;

monetary_tlr        : ’END’ ’LC_MONETARY’ EOL
                    ;

/* The following is the LC_NUMERIC category grammar */

lc_numeric          : numeric_hdr numeric_keywords       numeric_tlr
                    | numeric_hdr ’copy’ locale_name EOL numeric_tlr
                    ;

numeric_hdr         : ’LC_NUMERIC’ EOL
                    ;

numeric_keywords    : numeric_keywords numeric_keyword
                    | numeric_keyword
                    ;

numeric_keyword     : num_keyword_string num_string EOL
                    | num_keyword_grouping num_group_list EOL
                    ;

num_keyword_string  : ’decimal_point’
                    | ’thousands_sep’
                    ;

num_string          : ’"’ char_list ’"’
                    | ’""’
                    ;

num_keyword_grouping: ’grouping’
                    ;

num_group_list      : NUMBER
                    | num_group_list ’;’ NUMBER
                    ;

numeric_tlr         : ’END’ ’LC_NUMERIC’ EOL
                    ;

/* The following is the LC_TIME category grammar */

lc_time             : time_hdr time_keywords          time_tlr
                    | time_hdr ’copy’ locale_name EOL time_tlr
                    ;

time_hdr            : ’LC_TIME’ EOL
                    ;
```

```
        time_keywords       : time_keywords time_keyword
                            | time_keyword
                            ;

        time_keyword        : time_keyword_name time_list EOL
                            | time_keyword_fmt time_string EOL
                            | time_keyword_opt time_list EOL
                            ;

        time_keyword_name   : 'abday' | 'day' | 'abmon' | 'mon'
                            ;

        time_keyword_fmt    : 'd_t_fmt' | 'd_fmt' | 't_fmt'
                            | 'am_pm' | 't_fmt_ampm'
                            ;

        time_keyword_opt    : 'era' | 'era_d_fmt' | 'era_t_fmt'
                            | 'era_d_t_fmt' | 'alt_digits'
                            ;

        time_list           : time_list ';' time_string
                            | time_string
                            ;

        time_string         : '"' char_list '"'
                            ;

        time_tlr            : 'END' 'LC_TIME' EOL
                            ;
```

Chapter 8

Environment Variables

8.1 Environment Variable Definition

Environment variables defined in this chapter affect the operation of multiple utilities, functions, and applications. There are other environment variables that are of interest only to specific utilities. Environment variables that apply to a single utility only are defined as part of the utility description. See the ENVIRONMENT VARIABLES section of the utility descriptions in the Shell and Utilities volume of IEEE Std 1003.1-200x for information on environment variable usage.

The value of an environment variable is a string of characters. For a C-language program, an array of strings called the environment shall be made available when a process begins. The array is pointed to by the external variable *environ*, which is defined as:

```
extern char **environ;
```

These strings have the form *name*=*value*; *name*s shall not contain the character `'='`. For values to be portable across systems conforming to IEEE Std 1003.1-200x, the value shall be composed of characters from the portable character set (except NUL and as indicated below). There is no meaning associated with the order of strings in the environment. If more than one string in a process' environment has the same *name*, the consequences are undefined.

Environment variable names used by the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x consist solely of uppercase letters, digits, and the `'_'` (underscore) from the characters defined in Table 6-1 (on page 111) and do not begin with a digit. Other characters may be permitted by an implementation; applications shall tolerate the presence of such names. Uppercase and lowercase letters shall retain their unique identities and shall not be folded together. The name space of environment variable names containing lowercase letters is reserved for applications. Applications can define any environment variables with names from this name space without modifying the behavior of the standard utilities.

Note: Other applications may have difficulty dealing with environment variable names that start with a digit. For this reason, use of such names is not recommended anywhere.

The *values* that the environment variables may be assigned are not restricted except that they are considered to end with a null byte and the total space used to store the environment and the arguments to the process is limited to {ARG_MAX} bytes.

Other *name*=*value* pairs may be placed in the environment by, for example, calling any of the XSI *setenv*( ), *unsetenv*( ), or *putenv*( ) functions, manipulating the *environ* variable, or by using *envp* arguments when creating a process; see *exec* in the System Interfaces volume of IEEE Std 1003.1-200x.

It is unwise to conflict with certain variables that are frequently exported by widely used command interpreters and applications:

ARFLAGS
CC
CDPATH
CFLAGS
CHARSET
COLUMNS
DATEMSK
DEAD
EDITOR
ENV
EXINIT
FC
FCEDIT
FFLAGS
GET
GFLAGS
HISTFILE
HISTORY
HISTSIZE
HOME
IFS
LANG
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME
LDFLAGS
LEX
LFLAGS
LINENO
LINES
LISTER
LOGNAME
LPDEST
MAIL
MAILCHECK
MAILER
MAILPATH
MAILRC
MAKEFLAGS
MAKESHELL
MANPATH
MBOX
MORE
MSGVERB
NLSPATH
NPROC
OLDPWD
OPTARG
OPTERR
OPTIND
PAGER
PATH
PPID
PRINTER
PROCLANG
PROJECTDIR
PS1
PS2
PS3
PS4
PWD
RANDOM
SECONDS
SHELL
TERM
TERMCAP
TERMINFO
TMPDIR
TZ
USER
VISUAL
YACC
YFLAGS

If the variables in the following two sections are present in the environment during the execution of an application or utility, they shall be given the meaning described below. Some are placed into the environment by the implementation at the time the user logs in; all can be added or changed by the user or any ancestor of the current process. The implementation adds or changes environment variables named in IEEE Std 1003.1-200x only as specified in IEEE Std 1003.1-200x. If they are defined in the application's environment, the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x and the functions in the System Interfaces volume of IEEE Std 1003.1-200x assume they have the specified meaning. Conforming applications shall not set these environment variables to have meanings other than as described. See *getenv*( ) and the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.12, Shell Execution Environment for methods of accessing these variables.

8.2 Internationalization Variables

This section describes environment variables that are relevant to the operation of internationalized interfaces described in IEEE Std 1003.1-200x.

Users may use the following environment variables to announce specific localization requirements to applications. Applications can retrieve this information using the *setlocale*( ) function to initialize the correct behavior of the internationalized interfaces. The descriptions of the internationalization environment variables describe the resulting behavior only when the application locale is initialized in this way. The use of the internationalization variables by utilities described in the Shell and Utilities volume of IEEE Std 1003.1-200x are described in the ENVIRONMENT VARIABLES section for those utilities in addition to the global effects described in this section.

LANG This variable shall determine the locale category for native language, local customs, and coded character set in the absence of the *LC_ALL* and other *LC_** (*LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*) environment variables. This can be used by applications to determine the language to use for error messages and instructions, collating sequences, date formats, and so on.

LC_ALL	This variable shall determine the values for all locale categories. The value of the *LC_ALL* environment variable has precedence over any of the other environment variables starting with *LC_*(*LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*) and the *LANG* environment variable.
LC_COLLATE	This variable shall determine the locale category for character collation. It determines collation information for regular expressions and sorting, including equivalence classes and multi-character collating elements, in various utilities and the *strcoll*( ) and *strxfrm*( ) functions. Additional semantics of this variable, if any, are implementation-defined.
LC_CTYPE	This variable shall determine the locale category for character handling functions, such as *tolower*( ), *toupper*( ), and *isalpha*( ). This environment variable determines the interpretation of sequences of bytes of text data as characters (for example, single as opposed to multi-byte characters), the classification of characters (for example, alpha, digit, graph), and the behavior of character classes. Additional semantics of this variable, if any, are implementation-defined.
LC_MESSAGES	This variable shall determine the locale category for processing affirmative and negative responses and the language and cultural conventions in which messages should be written. It also affects the behavior of the *catopen*( ) function in determining the message catalog. Additional semantics of this variable, if any, are implementation-defined. The language and cultural conventions of diagnostic and informative messages whose format is unspecified by IEEE Std 1003.1-200x should be affected by the setting of *LC_MESSAGES*.
LC_MONETARY	This variable shall determine the locale category for monetary-related numeric formatting information. Additional semantics of this variable, if any, are implementation-defined.
LC_NUMERIC	This variable shall determine the locale category for numeric formatting (for example, thousands separator and radix character) information in various utilities as well as the formatted I/O operations in *printf*( ) and *scanf*( ) and the string conversion functions in *strtod*( ). Additional semantics of this variable, if any, are implementation-defined.
LC_TIME	This variable shall determine the locale category for date and time formatting information. It affects the behavior of the time functions in *strftime*( ). Additional semantics of this variable, if any, are implementation-defined.
NLSPATH	This variable shall contain a sequence of templates that the *catopen*( ) function uses when attempting to locate message catalogs. Each template consists of an optional prefix, one or more conversion specifications, a filename, and an optional suffix.

For example:

```
NLSPATH="/system/nlslib/%N.cat"
```

defines that *catopen*( ) should look for all message catalogs in the directory **/system/nlslib**, where the catalog name should be constructed from the *name* parameter passed to *catopen*( ) (%N), with the suffix **.cat**.

Conversion specifications consist of a ′%′ symbol, followed by a single-letter keyword. The following keywords are currently defined:

`%N`	The value of the *name* parameter passed to *catopen*( ).
`%L`	The value of the *LC_MESSAGES* category.
`%l`	The *language* element from the *LC_MESSAGES* category.
`%t`	The *territory* element from the *LC_MESSAGES* category.
`%c`	The *codeset* element from the *LC_MESSAGES* category.
`%%`	A single `'%'` character.

An empty string is substituted if the specified value is not currently defined. The separators underscore (`'_'`) and period (`'.'`) are not included in the `%t` and `%c` conversion specifications.

Templates defined in *NLSPATH* are separated by colons (`':'`). A leading or two adjacent colons `"::"` is equivalent to specifying `%N`. For example:

```
NLSPATH=":%N.cat:/nlslib/%L/%N.cat"
```

indicates to *catopen*( ) that it should look for the requested message catalog in *name*, *name*.**cat**, and **/nlslib/***category*/*name*.**cat**, where *category* is the value of the *LC_MESSAGES* category of the current locale.

Users should not set the *NLSPATH* variable unless they have a specific reason to override the default system path. Setting *NLSPATH* to override the default system path produces undefined results in the standard utilities and in applications with appropriate privileges.

The environment variables *LANG*, *LC_ALL*, *LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*, and *NLSPATH* provide for the support of internationalized applications. The standard utilities shall make use of these environment variables as described in this section and the individual ENVIRONMENT VARIABLES sections for the utilities. If these variables specify locale categories that are not based upon the same underlying codeset, the results are unspecified.

The values of locale categories shall be determined by a precedence order; the first condition met below determines the value:

1. If the *LC_ALL* environment variable is defined and is not null, the value of *LC_ALL* shall be used.
2. If the *LC_** environment variable (*LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME*) is defined and is not null, the value of the environment variable shall be used to initialize the category that corresponds to the environment variable.
3. If the *LANG* environment variable is defined and is not null, the value of the *LANG* environment variable shall be used.
4. If the *LANG* environment variable is not set or is set to the empty string, the implementation-defined default locale shall be used.

If the locale value is `"C"` or `"POSIX"`, the POSIX locale shall be used and the standard utilities behave in accordance with the rules in Section 7.2 (on page 120) for the associated category.

If the locale value begins with a slash, it shall be interpreted as the pathname of a file that was created in the output format used by the *localedef* utility; see OUTPUT FILES under *localedef*. Referencing such a pathname shall result in that locale being used for the indicated category.

XSI If the locale value has the form:

```
language[_territory][.codeset]
```

it refers to an implementation-provided locale, where settings of language, territory, and codeset are implementation-defined.

LC_COLLATE, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, and *LC_TIME* are defined to accept an additional field *@modifier*, which allows the user to select a specific instance of localization data within a single category (for example, for selecting the dictionary as opposed to the character ordering of data). The syntax for these environment variables is thus defined as:

```
[language[_territory][.codeset][@modifier]]
```

For example, if a user wanted to interact with the system in French, but required to sort German text files, *LANG* and *LC_COLLATE* could be defined as:

```
LANG=Fr_FR
LC_COLLATE=De_DE
```

This could be extended to select dictionary collation (say) by use of the *@modifier* field; for example:

```
LC_COLLATE=De_DE@dict
```

An implementation may support other formats.

If the locale value is not recognized by the implementation, the behavior is unspecified.

At runtime, these values are bound to a program's locale by calling the *setlocale*( ) function.

Additional criteria for determining a valid locale name are implementation-defined.

8.3 Other Environment Variables

COLUMNS	This variable shall represent a decimal integer >0 used to indicate the user's preferred width in column positions for the terminal screen or window; see Section 3.103 (on page 47). If this variable is unset or null, the implementation determines the number of columns, appropriate for the terminal or window, in an unspecified manner. When *COLUMNS* is set, any terminal-width information implied by *TERM* is overridden. Users and conforming applications should not set *COLUMNS* unless they wish to override the system selection and produce output unrelated to the terminal characteristics. Users should not need to set this variable in the environment unless there is a specific reason to override the implementation's default behavior, such as to display data in an area arbitrarily smaller than the terminal or window.
XSI *DATEMSK*	Indicates the pathname of the template file used by *getdate*( ).
HOME	The system shall initialize this variable at the time of login to be a pathname of the user's home directory. See **<pwd.h>**.
LINES	This variable shall represent a decimal integer >0 used to indicate the user's preferred number of lines on a page or the vertical screen or window size in lines. A line in this case is a vertical measure large enough to hold the tallest character in the character set being displayed. If this variable is unset or null, the implementation determines the number of lines, appropriate for the

		terminal or window (size, terminal baud rate, and so on), in an unspecified manner. When *LINES* is set, any terminal-height information implied by *TERM* is overridden. Users and conforming applications should not set *LINES* unless they wish to override the system selection and produce output unrelated to the terminal characteristics. Users should not need to set this variable in the environment unless there is a specific reason to override the implementation's default behavior, such as to display data in an area arbitrarily smaller than the terminal or window.
	LOGNAME	The system shall initialize this variable at the time of login to be the user's login name. See **<pwd.h>**. For a value of *LOGNAME* to be portable across implementations of IEEE Std 1003.1-200x, the value should be composed of characters from the portable filename character set.
XSI	*MSGVERB*	Describes which message components shall be used in writing messages by *fmtmsg*( ).
	PATH	This variable shall represent the sequence of path prefixes that certain functions and utilities apply in searching for an executable file known only by a filename. The prefixes shall be separated by a colon (`':'`). When a non-zero-length prefix is applied to this filename, a slash shall be inserted between the prefix and the filename. A zero-length prefix is a legacy feature that indicates the current working directory. It appears as two adjacent colons (`"::"`), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. A strictly conforming application shall use an actual pathname (such as **.**) to represent the current working directory in *PATH*. The list shall be searched from beginning to end, applying the filename to each prefix, until an executable file with the specified name and appropriate execution permissions is found. If the pathname being sought contains a slash, the search through the path prefixes shall not be performed. If the pathname begins with a slash, the specified path is resolved (see Section 4.11 (on page 98)). If *PATH* is unset or is set to null, the path search is implementation-defined.
	PWD	This variable shall represent an absolute pathname of the current working directory. It shall not contain any filename components of dot or dot-dot. The value is set by the *cd* utility.
	SHELL	This variable shall represent a pathname of the user's preferred command language interpreter. If this interpreter does not conform to the Shell Command Language in the Shell and Utilities volume of IEEE Std 1003.1-200x, Chapter 2, Shell Command Language, utilities may behave differently from those described in IEEE Std 1003.1-200x.
	TMPDIR	This variable shall represent a pathname of a directory made available for programs that need a place to create temporary files.
	TERM	This variable shall represent the terminal type for which output is to be prepared. This information is used by utilities and application programs wishing to exploit special capabilities specific to a terminal. The format and allowable values of this environment variable are unspecified.
	TZ	This variable shall represent timezone information. The contents of the environment variable named *TZ* shall be used by the *ctime*( ), *localtime*( ), *strftime*( ), and *mktime*( ) functions, and by various utilities, to override the default timezone. The value of *TZ* has one of the two forms (spaces inserted

for clarity):

`:characters`

or:

`std offset dst offset, rule`

If *TZ* is of the first format (that is, if the first character is a colon), the characters following the colon are handled in an implementation-defined manner.

The expanded format (for all *TZ*s whose value does not have a colon as the first character) is as follows:

`stdoffset[dst[offset][,start[/time],end[/time]]]`

Where:

std and *dst* Indicate no less than three, nor more than {TZNAME_MAX}, bytes that are the designation for the standard (*std*) or the alternative (*dst*—such as Daylight Savings Time) timezone. Only *std* is required; if *dst* is missing, then the alternative time does not apply in this locale.

Each of these fields may occur in either of two formats quoted or unquoted:

- In the quoted form, the first character shall be the less-than (`'<'`) character and the last character shall be the greater-than (`'>'`) character. All characters between these quoting characters shall be alphanumeric characters in the current locale, the plus-sign (`'+'`) character, or the minus-sign (`'−'`) character. The *std* and *dst* fields in this case shall not include the quoting characters. |
- In the unquoted form, all characters in these fields shall be alphabetic characters in the current locale.

The interpretation of these fields is unspecified if either field is less than three bytes (except for the case when *dst* is missing), more than {TZNAME_MAX} bytes, or if they contain characters other than those specified.

offset Indicates the value added to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

`hh[:mm[:ss]]`

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) shall be required and may be a single digit. The *offset* following *std* shall be required. If no *offset* follows *dst*, the alternative time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour shall be between zero and 24, and the minutes (and seconds)—if present—between zero and 59. The result of using values outside of this range is unspecified. If preceded by a `'−'`, the timezone shall be east of the Prime Meridian; otherwise, it shall be west (which may be indicated by an optional preceding `'+'`).

rule	Indicates when to change to and back from the alternative time. The *rule* has the form:

```
date[/time],date[/time]
```

where the first *date* describes when the change from standard to alternative time occurs and the second *date* describes when the change back happens. Each *time* field describes when, in current local time, the change to the other time is made.

The format of *date* is one of the following:

J*n*	The Julian day *n* ($1 \leq n \leq 365$). Leap days shall not be counted. That is, in all years—including leap years—February 28 is day 59 and March 1 is day 60. It is impossible to refer explicitly to the occasional February 29.
n	The zero-based Julian day ($0 \leq n \leq 365$). Leap days shall be counted, and it is possible to refer to February 29.
M*m.n.d*	The *d*'th day ($0 \leq d \leq 6$) of week *n* of month *m* of the year ($1 \leq n \leq 5$, $1 \leq m \leq 12$, where week 5 means "the last *d* day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*'th day occurs. Day zero is Sunday.

The *time* has the same format as *offset* except that no leading sign (`'−'` or `'+'`) is allowed. The default, if *time* is not given, shall be 02:00:00.

Chapter 9

Regular Expressions

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, where these character sets are interpreted according to the current locale. While many regular expressions can be interpreted differently depending on the current locale, many features, such as character class expressions, provide for contextual invariance across locales.

The Basic Regular Expression (BRE) notation and construction rules in Section 9.3 (on page 167) shall apply to most utilities supporting regular expressions. Some utilities, instead, support the Extended Regular Expressions (ERE) described in Section 9.4 (on page 171); any exceptions for both cases are noted in the descriptions of the specific utilities using regular expressions. Both BREs and EREs are supported by the Regular Expression Matching interface in the System Interfaces volume of IEEE Std 1003.1-200x under *regcomp*( ), *regexec*( ), and related functions.

9.1 Regular Expression Definitions

For the purposes of this section, the following definitions shall apply:

entire regular expression
The concatenated set of one or more BREs or EREs that make up the pattern specified for string selection.

matched
A sequence of zero or more characters shall be said to be matched by a BRE or ERE when the characters in the sequence correspond to a sequence of characters defined by the pattern.

Matching shall be based on the bit pattern used for encoding the character, not on the graphic representation of the character. This means that if a character set contains two or more encodings for a graphic symbol, or if the strings searched contain text encoded in more than one codeset, no attempt is made to search for any other representation of the encoded symbol. If that is required, the user can specify equivalence classes containing all variations of the desired graphic symbol.

The search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found, where *first* is defined to mean ‘‘begins earliest in the string’’. If the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched. For example: the BRE `"bb*"` matches the second to fourth characters of *abbbc*, and the ERE (*wee* | *week*)(*knights* | *night*) matches all ten characters of *weeknights*.

Consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string. For this purpose, a null string shall be considered to be longer than no match at all. For example, matching the BRE `"\(.*\).*"` against `"abcdef"`, the subexpression `"(\1)"` is `"abcdef"`, and matching the BRE `"\(a*\)*"` against `"bc"`, the subexpression `"(\1)"` is the null string.

When a multi-character collating element in a bracket expression (see Section 9.3.5 (on page 168)) is involved, the longest sequence shall be measured in characters consumed from the string to be matched; that is, the collating element counts not as one element, but as the number of characters it matches.

BRE (ERE) matching a single character

A BRE or ERE that shall match either a single character or a single collating element.

Only a BRE or ERE of this type that includes a bracket expression (see Section 9.3.5 (on page 168)) can match a collating element.

BRE (ERE) matching multiple characters

A BRE or ERE that shall match a concatenation of single characters or collating elements.

Such a BRE or ERE is made up from a BRE (ERE) matching a single character and BRE (ERE) special characters.

invalid

This section uses the term *invalid* for certain constructs or conditions. Invalid REs shall cause the utility or function using the RE to generate an error condition. When *invalid* is not used, violations of the specified syntax or semantics for REs produce undefined results: this may entail an error, enabling an extended syntax for that RE, or using the construct in error as literal characters to be matched. For example, the BRE construct `"\{1,2,3\}"` does not comply with the grammar. A conforming application cannot rely on it producing an error nor matching the literal characters `"\{1,2,3\}"`.

9.2 Regular Expression General Requirements

The requirements in this section shall apply to both basic and extended regular expressions.

The use of regular expressions is generally associated with text processing. REs (BREs and EREs) operate on text strings; that is, zero or more characters followed by an end-of-string delimiter (typically NUL). Some utilities employing regular expressions limit the processing to lines; that is, zero or more characters followed by a <newline>. In the regular expression processing described in IEEE Std 1003.1-200x, the <newline> is regarded as an ordinary character and both a period and a non-matching list can match one. The Shell and Utilities volume of IEEE Std 1003.1-200x specifies within the individual descriptions of those standard utilities employing regular expressions whether they permit matching of <newline>s; if not stated otherwise, the use of literal <newline>s or any escape sequence equivalent produces undefined results. Those utilities (like *grep*) that do not allow <newline>s to match are responsible for eliminating any <newline> from strings before matching against the RE. The *regcomp*( ) function in the System Interfaces volume of IEEE Std 1003.1-200x, however, can provide support for such processing without violating the rules of this section.

The interfaces specified in IEEE Std 1003.1-200x do not permit the inclusion of a NUL character in an RE or in the string to be matched. If during the operation of a standard utility a NUL is included in the text designated to be matched, that NUL may designate the end of the text string for the purposes of matching.

When a standard utility or function that uses regular expressions specifies that pattern matching shall be performed without regard to the case (uppercase or lowercase) of either data or patterns, then when each character in the string is matched against the pattern, not only the character, but also its case counterpart (if any), shall be matched. This definition of case-insensitive processing is intended to allow matching of multi-character collating elements as well as characters, as each character in the string is matched using both its cases. For example, in a locale where `"Ch"` is a multi-character collating element and where a matching list expression matches such elements, the RE `"[[.Ch.]]"` when matched against the string `"char"`, is in reality matched against `"ch"`, `"Ch"`, `"cH"`, and `"CH"`.

The implementation shall support any regular expression that does not exceed 256 bytes in length.

9.3 Basic Regular Expressions

9.3.1 BREs Matching a Single Character or Collating Element

A BRE ordinary character, a special character preceded by a backslash or a period, shall match a single character. A bracket expression shall match a single character or a single collating element.

9.3.2 BRE Ordinary Characters

An ordinary character is a BRE that matches itself: any character in the supported character set, except for the BRE special characters listed in Section 9.3.3.

The interpretation of an ordinary character preceded by a backslash (`'\'`) is undefined, except for:

- The characters `')'`, `'('`, `'{'`, and `'}'`
- The digits 1 to 9 inclusive (see Section 9.3.6 (on page 170))
- A character inside a bracket expression

9.3.3 BRE Special Characters

A *BRE special character* has special properties in certain contexts. Outside those contexts, or when preceded by a backslash, such a character is a BRE that matches the special character itself. The BRE special characters and the contexts in which they have their special meaning are as follows:

`. [ \`	The period, left-bracket, and backslash shall be special except when used in a bracket expression (see Section 9.3.5 (on page 168)). An expression containing a `'['` that is not preceded by a backslash and is not part of a bracket expression produces undefined results.
`*`	The asterisk shall be special except when used: • In a bracket expression • As the first character of an entire BRE (after an initial `'^'`, if any) • As the first character of a subexpression (after an initial `'^'`, if any); see Section 9.3.6 (on page 170)
`^`	The circumflex shall be special when used as: • An anchor (see Section 9.3.8 (on page 171)) • The first character of a bracket expression (see Section 9.3.5 (on page 168))
`$`	The dollar sign shall be special when used as an anchor.

9.3.4 Periods in BREs

A period (`'.'`), when used outside a bracket expression, is a BRE that shall match any character in the supported character set except NUL.

9.3.5 RE Bracket Expression

A bracket expression (an expression enclosed in square brackets, `"[ ]"`) is an RE that shall match a single collating element contained in the non-empty set of collating elements represented by the bracket expression.

The following rules and definitions apply to bracket expressions:

1. A *bracket expression* is either a matching list expression or a non-matching list expression. It consists of one or more expressions: collating elements, collating symbols, equivalence classes, character classes, or range expressions. The right-bracket (`']'`) shall lose its special meaning and represents itself in a bracket expression if it occurs first in the list (after an initial circumflex (`'^'`), if any). Otherwise, it shall terminate the bracket expression, unless it appears in a collating symbol (such as `"[.].]"`) or is the ending right-bracket for a collating symbol, equivalence class, or character class. The special characters `'.'`, `'*'`, `'['`, and `'\'` (period, asterisk, left-bracket, and backslash, respectively) shall lose their special meaning within a bracket expression.

   The character sequences `"[."`, `"[="`, and `"[:"` (left-bracket followed by a period, equals-sign, or colon) shall be special inside a bracket expression and are used to delimit collating symbols, equivalence class expressions, and character class expressions. These symbols shall be followed by a valid expression and the matching terminating sequence `".]"`, `"=]"`, or `":]"`, as described in the following items.

2. A *matching list expression* specifies a list that shall match any single-character collating element in any of the expressions represented in the list. The first character in the list shall not be the circumflex; for example, `"[abc]"` is an RE that matches any of the characters `'a'`, `'b'`, or `'c'`. It is unspecified whether a matching list expression matches a multi-character collating element that is matched by one of the expressions.

3. A *non-matching list expression* begins with a circumflex (`'^'`), and specifies a list that shall match any single-character collating element except for the expressions represented in the list after the leading circumflex. For example, `"[^abc]"` is an RE that matches any character except the characters `'a'`, `'b'`, or `'c'`. It is unspecified whether a non-matching list expression matches a multi-character collating element that is not matched by any of the expressions. The circumflex shall have this special meaning only when it occurs first in the list, immediately following the left-bracket.

4. A *collating symbol* is a collating element enclosed within bracket-period (`"[."` and `".]"`) delimiters. Collating elements are defined as described in Section 7.3.2.4 (on page 133). Conforming applications shall represent multi-character collating elements as collating symbols when it is necessary to distinguish them from a list of the individual characters that make up the multi-character collating element. For example, if the string `"ch"` is a collating element defined using the line:

   ```
   collating-element <ch-digraph> from "<c><h>"
   ```

   in the locale definition, the expression `"[[.ch.]]"` shall be treated as an RE containing the collating symbol `'ch'`, while `"[ch]"` shall be treated as an RE matching `'c'` or `'h'`. Collating symbols are recognized only inside bracket expressions. If the string is not a collating element in the current locale, the expression is invalid.

5. An *equivalence class expression* shall represent the set of collating elements belonging to an equivalence class, as described in Section 7.3.2.4 (on page 133). Only primary equivalence classes shall be recognized. The class shall be expressed by enclosing any one of the collating elements in the equivalence class within bracket-equal (`"[="` and `"=]"`) delimiters. For example, if `'a'`, `'à'`, and `'â'` belong to the same equivalence class, then `"[[=a=]b]"`, `"[[=à=]b]"`, and `"[[=â=]b]"` are each equivalent to `"[aàâb]"`. If the collating element does not belong to an equivalence class, the equivalence class expression shall be treated as a *collating symbol*.

6. A *character class expression* shall represent the union of two sets:

    a. The set of single-character collating elements whose characters belong to the character class, as defined in the *LC_CTYPE* category in the current locale.

    b. An unspecified set of multi-character collating elements.

    All character classes specified in the current locale shall be recognized. A character class expression is expressed as a character class name enclosed within bracket-colon (`"[:"` and `":]"`) delimiters.

    The following character class expressions shall be supported in all locales:

    ```
    [:alnum:]   [:cntrl:]   [:lower:]   [:space:]
    [:alpha:]   [:digit:]   [:print:]   [:upper:]
    [:blank:]   [:graph:]   [:punct:]   [:xdigit:]
    ```

    XSI In addition, character class expressions of the form:

    `[:`*`name`*`:]`

    are recognized in those locales where the *name* keyword has been given a **charclass** definition in the *LC_CTYPE* category.

7. In the POSIX locale, a range expression represents the set of collating elements that fall between two elements in the collation sequence, inclusive. In other locales, a range expression has unspecified behavior: strictly conforming applications shall not rely on whether the range expression is valid, or on the set of collating elements matched. A range expression shall be expressed as the starting point and the ending point separated by a hyphen (`'-'`).

    In the following, all examples assume the POSIX locale.

    The starting range point and the ending range point shall be a collating element or collating symbol. An equivalence class expression used as a starting or ending point of a range expression produces unspecified results. An equivalence class can be used portably within a bracket expression, but only outside the range. If the represented set of collating elements is empty, it is unspecified whether the expression matches nothing, or is treated as invalid.

    The interpretation of range expressions where the ending range point is also the starting range point of a subsequent range expression (for example, `"[a-m-o]"`) is undefined.

    The hyphen character shall be treated as itself if it occurs first (after an initial `'^'`, if any) or last in the list, or as an ending range point in a range expression. As examples, the expressions `"[-ac]"` and `"[ac-]"` are equivalent and match any of the characters `'a'`, `'c'`, or `'-'`; `"[^-ac]"` and `"[^ac-]"` are equivalent and match any characters except `'a'`, `'c'`, or `'-'`; the expression `"[%--]"` matches any of the characters between `'%'` and `'-'` inclusive; the expression `"[--@]"` matches any of the characters between `'-'` and `'@'` inclusive; and the expression `"[a--@]"` is either invalid or equivalent to `'@'`, because the letter `'a'` follows the symbol `'-'` in the POSIX locale. To use a hyphen as the starting range point, it shall either come first in the bracket expression or be specified as a collating symbol; for example, `"[][.-.]-0]"`, which matches either a right bracket or any character or collating element that collates between hyphen and 0, inclusive.

If a bracket expression specifies both `'-'` and `']'`, the `']'` shall be placed first (after the `'^'`, if any) and the `'-'` last within the bracket expression.

9.3.6 BREs Matching Multiple Characters

The following rules can be used to construct BREs matching multiple characters from BREs matching a single character:

1. The concatenation of BREs shall match the concatenation of the strings matched by each component of the BRE.

2. A *subexpression* can be defined within a BRE by enclosing it between the character pairs `"\("` and `"\)"`. Such a subexpression shall match whatever it would have matched without the `"\("` and `"\)"`, except that anchoring within subexpressions is optional behavior; see Section 9.3.8 (on page 171). Subexpressions can be arbitrarily nested.

3. The *back-reference expression* `'\n'` shall match the same (possibly empty) string of characters as was matched by a subexpression enclosed between `"\("` and `"\)"` preceding the `'\n'`. The character `'n'` shall be a digit from 1 through 9, specifying the *n*th subexpression (the one that begins with the *n*th `"\("` from the beginning of the pattern and ends with the corresponding paired `"\)"`). The expression is invalid if less than *n* subexpressions precede the `'\n'`. For example, the expression `"\(.*\)\1$"` matches a line consisting of two adjacent appearances of the same string, and the expression `"\(a\)*\1"` fails to match `'a'`. When the referenced subexpression matched more than one string, the back-referenced expression shall refer to the last matched string. If the subexpression referenced by the back-reference matches more than one string because of an asterisk (`'*'`) or an interval expression (see item (5)), the back-reference shall match the last (rightmost) of these strings.

4. When a BRE matching a single character, a subexpression, or a back-reference is followed by the special character asterisk (`'*'`), together with that asterisk it shall match what zero or more consecutive occurrences of the BRE would match. For example, `"[ab]*"` and `"[ab][ab]"` are equivalent when matching the string `"ab"`.

5. When a BRE matching a single character, a subexpression, or a back-reference is followed by an *interval expression* of the format `"\{m\}"`, `"\{m,\}"`, or `"\{m,n\}"`, together with that interval expression it shall match what repeated consecutive occurrences of the BRE would match. The values of *m* and *n* are decimal integers in the range $0 \le m \le n \le$ {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression `"\{m\}"` shall match exactly *m* occurrences of the preceding BRE, `"\{m,\}"` shall match at least *m* occurrences, and `"\{m,n\}"` shall match any number of occurrences between *m* and *n*, inclusive.

   For example, in the string `"abababccccccd"` the BRE `"c\{3\}"` is matched by characters `'7'` to `'9'`, the BRE `"\(ab\)\{4,\}"` is not matched at all, and the BRE `"c\{1,3\}d"` is matched by characters ten to thirteen.

The behavior of multiple adjacent duplication symbols (`'*'` and intervals) produces undefined results.

A subexpression repeated by an asterisk (`'*'`) or an interval expression shall not match a null expression unless this is the only match for the repetition or it is necessary to satisfy the exact or minimum number of occurrences for the interval expression.

9.3.7 BRE Precedence

The order of precedence shall be as shown in the following table:

BRE Precedence (from high to low)	
Collation-related bracket symbols	`[==] [::] [..]`
Escaped characters	`\<special character>`
Bracket expression	`[]`
Subexpressions / back-references	`\(\) \n`
Single-character-BRE duplication	`* \{m,n\}`
Concatenation	
Anchoring	`^ $`

9.3.8 BRE Expression Anchoring

A BRE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar sign special characters shall be considered BRE anchors in the following contexts:

1. A circumflex (`'^'`) shall be an anchor when used as the first character of an entire BRE. The implementation may treat the circumflex as an anchor when used as the first character of a subexpression. The circumflex shall anchor the expression (or optionally subexpression) to the beginning of a string; only sequences starting at the first character of a string shall be matched by the BRE. For example, the BRE `"^ab"` matches `"ab"` in the string `"abcdef"`, but fails to match in the string `"cdefab"`. The BRE `"\(^ab\)"` may match the former string. A portable BRE shall escape a leading circumflex in a subexpression to match a literal circumflex.

2. A dollar sign (`'$'`) shall be an anchor when used as the last character of an entire BRE. The implementation may treat a dollar sign as an anchor when used as the last character of a subexpression. The dollar sign shall anchor the expression (or optionally subexpression) to the end of the string being matched; the dollar sign can be said to match the end-of-string following the last character.

3. A BRE anchored by both `'^'` and `'$'` shall match only an entire string. For example, the BRE `"^abcdef$"` matches strings consisting only of `"abcdef"`.

9.4 Extended Regular Expressions

The *extended regular expression* (ERE) notation and construction rules shall apply to utilities defined as using extended regular expressions; any exceptions to the following rules are noted in the descriptions of the specific utilities using EREs.

9.4.1 EREs Matching a Single Character or Collating Element

An ERE ordinary character, a special character preceded by a backslash, or a period shall match a single character. A bracket expression shall match a single character or a single collating element. An *ERE matching a single character* enclosed in parentheses shall match the same as the ERE without parentheses would have matched.

9.4.2 ERE Ordinary Characters

An *ordinary character* is an ERE that matches itself. An ordinary character is any character in the supported character set, except for the ERE special characters listed in Section 9.4.3. The interpretation of an ordinary character preceded by a backslash (`'\'`) is undefined.

9.4.3 ERE Special Characters

An *ERE special character* has special properties in certain contexts. Outside those contexts, or when preceded by a backslash, such a character shall be an ERE that matches the special character itself. The extended regular expression special characters and the contexts in which they shall have their special meaning are as follows:

`. [ \ (` The period, left-bracket, backslash, and left-parenthesis shall be special except when used in a bracket expression (see Section 9.3.5 (on page 168)). Outside a bracket expression, a left-parenthesis immediately followed by a right-parenthesis produces undefined results.

`)` The right-parenthesis shall be special when matched with a preceding left-parenthesis, both outside a bracket expression.

`* + ? {` The asterisk, plus-sign, question-mark, and left-brace shall be special except when used in a bracket expression (see Section 9.3.5 (on page 168)). Any of the following uses produce undefined results:

- If these characters appear first in an ERE, or immediately following a vertical-line, circumflex, or left-parenthesis
- If a left-brace is not part of a valid interval expression (see Section 9.4.6 (on page 173))

`|` The vertical-line is special except when used in a bracket expression (see Section 9.3.5 (on page 168)). A vertical-line appearing first or last in an ERE, or immediately following a vertical-line or a left-parenthesis, or immediately preceding a right-parenthesis, produces undefined results.

`^` The circumflex shall be special when used as:

- An anchor (see Section 9.4.9 (on page 174))
- The first character of a bracket expression (see Section 9.3.5 (on page 168))

`$` The dollar sign shall be special when used as an anchor.

9.4.4 Periods in EREs

A period (`'.'`), when used outside a bracket expression, is an ERE that shall match any character in the supported character set except NUL.

9.4.5 ERE Bracket Expression

The rules for ERE Bracket Expressions are the same as for Basic Regular Expressions; see Section 9.3.5 (on page 168).

9.4.6 EREs Matching Multiple Characters

The following rules shall be used to construct EREs matching multiple characters from EREs matching a single character:

1. A *concatenation of EREs* shall match the concatenation of the character sequences matched by each component of the ERE. A concatenation of EREs enclosed in parentheses shall match whatever the concatenation without the parentheses matches. For example, both the ERE `"cd"` and the ERE `"(cd)"` are matched by the third and fourth character of the string `"abcdefabcdef"`.

2. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character plus-sign (`'+'`), together with that plus-sign it shall match what one or more consecutive occurrences of the ERE would match. For example, the ERE `"b+(bc)"` matches the fourth to seventh characters in the string `"acabbbcde"`. And, `"[ab]+"` and `"[ab][ab]*"` are equivalent.

3. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character asterisk (`'*'`), together with that asterisk it shall match what zero or more consecutive occurrences of the ERE would match. For example, the ERE `"b*c"` matches the first character in the string `"cabbbcde"`, and the ERE `"b*cd"` matches the third to seventh characters in the string `"cabbbcdebbbbbbcdbc"`. And, `"[ab]*"` and [ab][ab] are equivalent when matching the string `"ab"`.

4. When an ERE matching a single character or an ERE enclosed in parentheses is followed by the special character question-mark (`'?'`), together with that question-mark it shall match what zero or one consecutive occurrences of the ERE would match. For example, the ERE `"b?c"` matches the second character in the string `"acabbbcde"`.

5. When an ERE matching a single character or an ERE enclosed in parentheses is followed by an *interval expression* of the format `"{m}"`, `"{m,}"`, or `"{m,n}"`, together with that interval expression it shall match what repeated consecutive occurrences of the ERE would match. The values of *m* and *n* are decimal integers in the range $0 \leq m \leq n \leq$ {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression `"{m}"` matches exactly *m* occurrences of the preceding ERE, `"{m,}"` matches at least *m* occurrences, and `"{m,n}"` matches any number of occurrences between *m* and *n*, inclusive.

   For example, in the string `"abababccccccd"` the ERE `"c{3}"` is matched by characters `'7'` to `'9'` and the ERE `"(ab){2,}"` is matched by characters one to six.

The behavior of multiple adjacent duplication symbols (`'+'`, `'*'`, `'?'`, and intervals) produces undefined results.

An ERE matching a single character repeated by an `'*'`, `'?'`, or an interval expression shall not match a null expression unless this is the only match for the repetition or it is necessary to satisfy the exact or minimum number of occurrences for the interval expression.

9.4.7 ERE Alternation

Two EREs separated by the special character vertical-line (`'|'`) shall match a string that is matched by either. For example, the ERE `"a((bc)|d)"` matches the string `"abc"` and the string `"ad"`. Single characters, or expressions matching single characters, separated by the vertical bar and enclosed in parentheses, shall be treated as an ERE matching a single character.

9.4.8 ERE Precedence

The order of precedence shall be as shown in the following table:

ERE Precedence (from high to low)	
Collation-related bracket symbols	`[==] [::] [..]`
Escaped characters	`\<special character>`
Bracket expression	`[]`
Grouping	`()`
Single-character-ERE duplication	`* + ? {m,n}`
Concatenation	
Anchoring	`^ $`
Alternation	`|`

For example, the ERE `"abba|cde"` matches either the string `"abba"` or the string `"cde"` (rather than the string `"abbade"` or `"abbcde"`, because concatenation has a higher order of precedence than alternation).

9.4.9 ERE Expression Anchoring

An ERE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar sign special characters shall be considered ERE anchors when used anywhere outside a bracket expression. This shall have the following effects:

1. A circumflex (`'^'`) outside a bracket expression shall anchor the expression or subexpression it begins to the beginning of a string; such an expression or subexpression can match only a sequence starting at the first character of a string. For example, the EREs `"^ab"` and `"(^ab)"` match `"ab"` in the string `"abcdef"`, but fail to match in the string `"cdefab"`, and the ERE `"a^b"` is valid, but can never match because the `'a'` prevents the expression `"^b"` from matching starting at the first character.

2. A dollar sign (`'$'`) outside a bracket expression shall anchor the expression or subexpression it ends to the end of a string; such an expression or subexpression can match only a sequence ending at the last character of a string. For example, the EREs `"ef$"` and `"(ef$)"` match `"ef"` in the string `"abcdef"`, but fail to match in the string `"cdefab"`, and the ERE `"e$f"` is valid, but can never match because the `'f'` prevents the expression `"e$"` from matching ending at the last character.

9.5 Regular Expression Grammar

Grammars describing the syntax of both basic and extended regular expressions are presented in this section. The grammar takes precedence over the text. See the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 1.10, Grammar Conventions.

9.5.1 BRE/ERE Grammar Lexical Conventions

The lexical conventions for regular expressions are as described in this section.

Except as noted, the longest possible token or delimiter beginning at a given point is recognized.

The following tokens are processed (in addition to those string constants shown in the grammar):

COLL_ELEM_SINGLE	Any single-character collating element, unless it is a META_CHAR.
COLL_ELEM_MULTI	Any multi-character collating element.
BACKREF	Applicable only to basic regular expressions. The character string consisting of `'\'` followed by a single-digit numeral, `'1'` to `'9'`.
DUP_COUNT	Represents a numeric constant. It shall be an integer in the range 0 ≤DUP_COUNT ≤{RE_DUP_MAX}. This token is only recognized when the context of the grammar requires it. At all other times, digits not preceded by `'\'` are treated as ORD_CHAR.
META_CHAR	One of the characters: ^ When found first in a bracket expression – When found anywhere but first (after an initial `'^'`, if any) or last in a bracket expression, or as the ending range point in a range expression] When found anywhere but first (after an initial `'^'`, if any) in a bracket expression
L_ANCHOR	Applicable only to basic regular expressions. The character `'^'` when it appears as the first character of a basic regular expression and when not QUOTED_CHAR. The `'^'` may be recognized as an anchor elsewhere; see Section 9.3.8 (on page 171).
ORD_CHAR	A character, other than one of the special characters in SPEC_CHAR.

QUOTED_CHAR	In a BRE, one of the character sequences:
	`\^ \. \* \[ \$ \\`
	In an ERE, one of the character sequences:
	`\^ \. \[ \$ \( \) |`
	`\* \+ \? \{ \\`

R_ANCHOR	(Applicable only to basic regular expressions.) The character `'$'` when it appears as the last character of a basic regular expression and when not QUOTED_CHAR. The `'$'` may be recognized as an anchor elsewhere; see Section 9.3.8 (on page 171).
SPEC_CHAR	For basic regular expressions, one of the following special characters:

`.`	Anywhere outside bracket expressions
`\`	Anywhere outside bracket expressions
`[`	Anywhere outside bracket expressions
`^`	When used as an anchor (see Section 9.3.8 (on page 171)) or when first in a bracket expression
`$`	When used as an anchor
`*`	Anywhere except first in an entire RE, anywhere in a bracket expression, directly following `"\("`, directly following an anchoring `'^'`

For extended regular expressions, shall be one of the following special characters found anywhere outside bracket expressions:

```
^    .    [    $    (    )    |
*    +    ?    {    \
```

(The close-parenthesis shall be considered special in this context only if matched with a preceding open-parenthesis.)

9.5.2 RE and Bracket Expression Grammar

This section presents the grammar for basic regular expressions, including the bracket expression grammar that is common to both BREs and EREs.

```
%token    ORD_CHAR QUOTED_CHAR DUP_COUNT

%token    BACKREF L_ANCHOR R_ANCHOR

%token    Back_open_paren  Back_close_paren
/*          '\('             '\)'        */

%token    Back_open_brace  Back_close_brace
/*          '\{'             '\}'         */

/* The following tokens are for the Bracket Expression
   grammar common to both REs and EREs. */

%token    COLL_ELEM_SINGLE COLL_ELEM_MULTI META_CHAR

%token    Open_equal Equal_close Open_dot Dot_close Open_colon Colon_close
/*           '[='       '=]'        '[.'     '.]'      '[:'       ':]'  */

%token    class_name
/* class_name is a keyword to the LC_CTYPE locale category */
/* (representing a character class) in the current locale */
/* and is only recognized between [: and :] */

%start    basic_reg_exp
%%

/* --------------------------------------------
   Basic Regular Expression
   --------------------------------------------
*/
basic_reg_exp  :          RE_expression
               | L_ANCHOR
               |                       R_ANCHOR
```

```
               | L_ANCHOR               R_ANCHOR
               | L_ANCHOR RE_expression
               |          RE_expression R_ANCHOR
               | L_ANCHOR RE_expression R_ANCHOR
               ;
RE_expression  :               simple_RE
               | RE_expression simple_RE
               ;
simple_RE      : nondupl_RE
               | nondupl_RE RE_dupl_symbol
               ;
nondupl_RE     : one_char_or_coll_elem_RE
               | Back_open_paren RE_expression Back_close_paren
               | BACKREF
               ;
one_char_or_coll_elem_RE : ORD_CHAR
               | QUOTED_CHAR
               | '.'
               | bracket_expression
               ;
RE_dupl_symbol : '*'
               | Back_open_brace DUP_COUNT               Back_close_brace
               | Back_open_brace DUP_COUNT ','           Back_close_brace
               | Back_open_brace DUP_COUNT ',' DUP_COUNT Back_close_brace
               ;

/* --------------------------------------------
   Bracket Expression
   -------------------------------------------
*/
bracket_expression : '[' matching_list ']'
               | '[' nonmatching_list ']'
               ;
matching_list  : bracket_list
               ;
nonmatching_list : '^' bracket_list
               ;
bracket_list   : follow_list
               | follow_list '-'
               ;
follow_list    :             expression_term
               | follow_list expression_term
               ;
expression_term : single_expression
               | range_expression
               ;
single_expression : end_range
               | character_class
               | equivalence_class
               ;
range_expression : start_range end_range
               | start_range '-'
               ;
```

```
start_range    : end_range ’-’
               ;
end_range      : COLL_ELEM_SINGLE
               | collating_symbol
               ;
collating_symbol : Open_dot COLL_ELEM_SINGLE Dot_close
               | Open_dot COLL_ELEM_MULTI Dot_close
               | Open_dot META_CHAR Dot_close
               ;
equivalence_class : Open_equal COLL_ELEM_SINGLE Equal_close
               | Open_equal COLL_ELEM_MULTI Equal_close
               ;
character_class : Open_colon class_name Colon_close
               ;
```

The BRE grammar does not permit L_ANCHOR or R_ANCHOR inside "\(" and "\)" (which implies that ’ˆ’ and ’$’ are ordinary characters). This reflects the semantic limits on the application, as noted in Section 9.3.8 (on page 171). Implementations are permitted to extend the language to interpret ’ˆ’ and ’$’ as anchors in these locations, and as such, conforming applications cannot use unescaped ’ˆ’ and ’$’ in positions inside "\(" and "\)" that might be interpreted as anchors.

9.5.3 ERE Grammar

This section presents the grammar for extended regular expressions, excluding the bracket expression grammar.

Note: The bracket expression grammar and the associated **%token** lines are identical between BREs and EREs. It has been omitted from the ERE section to avoid unnecessary editorial duplication.

```
%token  ORD_CHAR QUOTED_CHAR DUP_COUNT
%start  extended_reg_exp
%%

/* --------------------------------------------
   Extended Regular Expression
   --------------------------------------------
*/
extended_reg_exp   :                          ERE_branch
                   | extended_reg_exp ’|’ ERE_branch
                   ;
ERE_branch         :              ERE_expression
                   | ERE_branch ERE_expression
                   ;
ERE_expression     : one_char_or_coll_elem_ERE
                   | ’ˆ’
                   | ’$’
                   | ’(’ extended_reg_exp ’)’
                   | ERE_expression ERE_dupl_symbol
                   ;
one_char_or_coll_elem_ERE  : ORD_CHAR
                   | QUOTED_CHAR
                   | ’.’
                   | bracket_expression
                   ;
```

```
ERE_dupl_symbol    : '*'
                   | '+'
                   | '?'
                   | '{' DUP_COUNT               '}'
                   | '{' DUP_COUNT ','           '}'
                   | '{' DUP_COUNT ',' DUP_COUNT '}'
                   ;
```

The ERE grammar does not permit several constructs that previous sections specify as having undefined results:

- ORD_CHAR preceded by `'\'`
- One or more *ERE_dupl_symbol*s appearing first in an ERE, or immediately following `'|'`, `'^'`, or `'('`
- `'{'` not part of a valid *ERE_dupl_symbol*
- `'|'` appearing first or last in an ERE, or immediately following `'|'` or `'('`, or immediately preceding `')'`

Implementations are permitted to extend the language to allow these. Conforming applications cannot use such constructs. |

Chapter 10

Directory Structure and Devices

10.1 Directory Structure and Files

The following directories shall exist on conforming systems and conforming applications shall make use of them only as described. Strictly conforming applications shall not assume the ability to create files in any of these directories, unless specified below.

/ The root directory.

/dev Contains **/dev/console**, **/dev/null**, and **/dev/tty**, described below.

The following directory shall exist on conforming systems and shall be used as described.

/tmp A directory made available for programs that need a place to create temporary files. Applications shall be allowed to create files in this directory, but shall not assume that such files are preserved between invocations of the application.

The following files shall exist on conforming systems and shall be both readable and writable.

/dev/null An infinite data source and data sink. Data written to **/dev/null** shall be discarded. Reads from **/dev/null** shall always return end-of-file (EOF).

/dev/tty In each process, a synonym for the controlling terminal associated with the process group of that process, if any. It is useful for programs or shell procedures that wish to be sure of writing messages to or reading data from the terminal no matter how output has been redirected. It can also be used for programs that demand the name of a file for output, when typed output is desired and it is tiresome to find out what terminal is currently in use.

The following file shall exist on conforming systems and need not be readable or writable:

/dev/console The **/dev/console** file is a generic name given to the system console (see Section 3.382 (on page 85)). It is usually linked to an implementation-defined special file. It shall provide an interface to the system console conforming to the requirements of the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

10.2 Output Devices and Terminal Types

The utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x historically have been implemented on a wide range of terminal types, but a conforming implementation need not support all features of all utilities on every conceivable terminal. IEEE Std 1003.1-200x states which features are optional for certain classes of terminals in the individual utility description sections. The implementation shall document which terminal types it supports and which of these features and utilities are not supported by each terminal.

When a feature or utility is not supported on a specific terminal type, as allowed by IEEE Std 1003.1-200x, and the implementation considers such a condition to be an error preventing use of the feature or utility, the implementation shall indicate such conditions through diagnostic messages or exit status values or both (as appropriate to the specific utility description) that inform the user that the terminal type lacks the appropriate capability.

IEEE Std 1003.1-200x uses a notational convention based on historical practice that identifies some of the control characters defined in Section 7.3.1 (on page 122) in a manner easily remembered by users on many terminals. The correspondence between this ‘‘<control>-*char*’’ notation and the actual control characters is shown in the following table. When IEEE Std 1003.1-200x refers to a character by its <control>- *name*, it is referring to the actual control character shown in the Value column of the table, which is not necessarily the exact control key sequence on all terminals. Some terminals have keyboards that do not allow the direct transmission of all the non-alphanumeric characters shown. In such cases, the system documentation shall describe which data sequences transmitted by the terminal are interpreted by the system as representing the special characters.

Table 10-1 Control Character Names

Name	Value	Symbolic Name	Name	Value	Symbolic Name
<control>-A	<SOH>	<SOH>	<control>-Q	<DC1>	<DC1>
<control>-B	<STX>	<STX>	<control>-R	<DC2>	<DC2>
<control>-C	<ETX>	<ETX>	<control>-S	<DC3>	<DC3>
<control>-D	<EOT>	<EOT>	<control>-T	<DC4>	<DC4>
<control>-E	<ENQ>	<ENQ>	<control>-U	<NAK>	<NAK>
<control>-F	<ACK>	<ACK>	<control>-V	<SYN>	<SYN>
<control>-G	<BEL>	<alert>	<control>-W	<ETB>	<ETB>
<control>-H	<BS>	<backspace>	<control>-X	<CAN>	<CAN>
<control>-I	<HT>	<tab>	<control>-Y	<EM>	<EM>
<control>-J	<LF>	<linefeed>	<control>-Z	<SUB>	<SUB>
<control>-K	<VT>	<vertical-tab>	<control>-[	<ESC>	<ESC>
<control>-L	<FF>	<form-feed>	<control>-\	<FS>	<FS>
<control>-M	<CR>	<carriage-return>	<control>-]	<GS>	<GS>
<control>-N	<SO>	<SO>	<control>-^	<RS>	<RS>
<control>-O	<SI>	<SI>	<control>-_	<US>	<US>
<control>-P	<DLE>	<DLE>	<control>-?	<DEL>	<DEL>

Note: The notation uses uppercase letters for arbitrary editorial reasons. There is no implication that the keystrokes represent control-shift-letter sequences.

Chapter 11

General Terminal Interface

This chapter describes a general terminal interface that shall be provided. It shall be supported on any asynchronous communications ports if the implementation provides them. It is implementation-defined whether it supports network connections or synchronous ports, or both.

11.1 Interface Characteristics

11.1.1 Opening a Terminal Device File

When a terminal device file is opened, it normally causes the thread to wait until a connection is established. In practice, application programs seldom open these files; they are opened by special programs and become an application's standard input, output, and error files.

As described in *open*( ), opening a terminal device file with the O_NONBLOCK flag clear shall cause the thread to block until the terminal device is ready and available. If CLOCAL mode is not set, this means blocking until a connection is established. If CLOCAL mode is set in the terminal, or the O_NONBLOCK flag is specified in the *open*( ), the *open*( ) function shall return a file descriptor without waiting for a connection to be established.

11.1.2 Process Groups

A terminal may have a foreground process group associated with it. This foreground process group plays a special role in handling signal-generating input characters, as discussed in Section 11.1.9 (on page 187).

A command interpreter process supporting job control can allocate the terminal to different jobs, or process groups, by placing related processes in a single process group and associating this process group with the terminal. A terminal's foreground process group may be set or examined by a process, assuming the permission requirements are met; see *tcgetpgrp*( ) and *tcsetpgrp*( ). The terminal interface aids in this allocation by restricting access to the terminal by processes that are not in the current process group; see Section 11.1.4 (on page 184).

When there is no longer any process whose process ID or process group ID matches the process group ID of the foreground process group, the terminal shall have no foreground process group. It is unspecified whether the terminal has a foreground process group when there is a process whose process ID matches the foreground process ID, but whose process group ID does not. No actions defined in IEEE Std 1003.1-200x, other than allocation of a controlling terminal or a successful call to *tcsetpgrp*( ), cause a process group to become the foreground process group of the terminal.

11.1.3 The Controlling Terminal

A terminal may belong to a process as its controlling terminal. Each process of a session that has a controlling terminal has the same controlling terminal. A terminal may be the controlling terminal for at most one session. The controlling terminal for a session is allocated by the session leader in an implementation-defined manner. If a session leader has no controlling terminal, and opens a terminal device file that is not already associated with a session without using the O_NOCTTY option (see *open*( )), it is implementation-defined whether the terminal becomes the controlling terminal of the session leader. If a process which is not a session leader opens a terminal file, or the O_NOCTTY option is used on *open*( ), then that terminal shall not become the controlling terminal of the calling process. When a controlling terminal becomes associated with a session, its foreground process group shall be set to the process group of the session leader.

The controlling terminal is inherited by a child process during a *fork*( ) function call. A process relinquishes its controlling terminal when it creates a new session with the *setsid*( ) function; other processes remaining in the old session that had this terminal as their controlling terminal continue to have it. Upon the close of the last file descriptor in the system (whether or not it is in the current session) associated with the controlling terminal, it is unspecified whether all processes that had that terminal as their controlling terminal cease to have any controlling terminal. Whether and how a session leader can reacquire a controlling terminal after the controlling terminal has been relinquished in this fashion is unspecified. A process does not relinquish its controlling terminal simply by closing all of its file descriptors associated with the controlling terminal if other processes continue to have it open.

When a controlling process terminates, the controlling terminal is dissociated from the current session, allowing it to be acquired by a new session leader. Subsequent access to the terminal by other processes in the earlier session may be denied, with attempts to access the terminal treated as if a modem disconnect had been sensed.

11.1.4 Terminal Access Control

If a process is in the foreground process group of its controlling terminal, read operations shall be allowed, as described in Section 11.1.5 (on page 185). Any attempts by a process in a background process group to read from its controlling terminal cause its process group to be sent a SIGTTIN signal unless one of the following special cases applies: if the reading process is ignoring or blocking the SIGTTIN signal, or if the process group of the reading process is orphaned, the *read*( ) shall return −1, with *errno* set to [EIO] and no signal shall be sent. The default action of the SIGTTIN signal shall be to stop the process to which it is sent. See **<signal.h>**.

If a process is in the foreground process group of its controlling terminal, write operations shall be allowed as described in Section 11.1.8 (on page 187). Attempts by a process in a background process group to write to its controlling terminal shall cause the process group to be sent a SIGTTOU signal unless one of the following special cases applies: if TOSTOP is not set, or if TOSTOP is set and the process is ignoring or blocking the SIGTTOU signal, the process is allowed to write to the terminal and the SIGTTOU signal is not sent. If TOSTOP is set, and the process group of the writing process is orphaned, and the writing process is not ignoring or blocking the SIGTTOU signal, the *write*( ) shall return −1, with *errno* set to [EIO] and no signal shall be sent.

Certain calls that set terminal parameters are treated in the same fashion as *write*( ), except that TOSTOP is ignored; that is, the effect is identical to that of terminal writes when TOSTOP is set (see Section 11.2.5 (on page 193), *tcdrain*( ), *tcflow*( ), *tcflush*( ), *tcsendbreak*( ), *tcsetattr*( ), and *tcsetpgrp*( )).

11.1.5 Input Processing and Reading Data

A terminal device associated with a terminal device file may operate in full-duplex mode, so that data may arrive even while output is occurring. Each terminal device file has an *input queue*, associated with it, into which incoming data is stored by the system before being read by a process. The system may impose a limit, {MAX_INPUT}, on the number of bytes that may be stored in the input queue. The behavior of the system when this limit is exceeded is implementation-defined.

Two general kinds of input processing are available, determined by whether the terminal device file is in canonical mode or non-canonical mode. These modes are described in Section 11.1.6 and Section 11.1.7 (on page 186). Additionally, input characters are processed according to the **c_iflag** (see Section 11.2.2 (on page 189)) and **c_lflag** (see Section 11.2.5 (on page 193)) fields. Such processing can include *echoing*, which in general means transmitting input characters immediately back to the terminal when they are received from the terminal. This is useful for terminals that can operate in full-duplex mode.

The manner in which data is provided to a process reading from a terminal device file is dependent on whether the terminal file is in canonical or non-canonical mode, and on whether or not the O_NONBLOCK flag is set by *open*( ) or *fcntl*( ).

If the O_NONBLOCK flag is clear, then the read request shall be blocked until data is available or a signal has been received. If the O_NONBLOCK flag is set, then the read request shall be completed, without blocking, in one of three ways:

1. If there is enough data available to satisfy the entire request, the *read*( ) shall complete successfully and shall return the number of bytes read.

2. If there is not enough data available to satisfy the entire request, the *read*( ) shall complete successfully, having read as much data as possible, and shall return the number of bytes it was able to read.

3. If there is no data available, the *read*( ) shall return –1, with *errno* set to [EAGAIN].

When data is available depends on whether the input processing mode is canonical or non-canonical. The following sections, Section 11.1.6 and Section 11.1.7 (on page 186), describe each of these input processing modes.

11.1.6 Canonical Mode Input Processing

In canonical mode input processing, terminal input is processed in units of lines. A line is delimited by a newline character (NL), an end-of-file character (EOF), or an end-of-line (EOL) character. See Section 11.1.9 (on page 187) for more information on EOF and EOL. This means that a read request shall not return until an entire line has been typed or a signal has been received. Also, no matter how many bytes are requested in the *read*( ) call, at most one line shall be returned. It is not, however, necessary to read a whole line at once; any number of bytes, even one, may be requested in a *read*( ) without losing information.

If {MAX_CANON} is defined for this terminal device, it shall be a limit on the number of bytes in a line. The behavior of the system when this limit is exceeded is implementation-defined. If {MAX_CANON} is not defined, there shall be no such limit; see *pathconf*( ).

Erase and kill processing occur when either of two special characters, the ERASE and KILL characters (see Section 11.1.9 (on page 187)), is received. This processing shall affect data in the input queue that has not yet been delimited by a newline (NL), EOF, or EOL character. This undelimited data makes up the current line. The ERASE character shall delete the last character in the current line, if there is one. The KILL character shall delete all data in the current line, if there are any. The ERASE and KILL characters shall have no effect if there is no data in the current line. The ERASE and KILL characters themselves shall not be placed in the input queue.

11.1.7 Non-Canonical Mode Input Processing

In non-canonical mode input processing, input bytes are not assembled into lines, and erase and kill processing shall not occur. The values of the MIN and TIME members of the **c_cc** array are used to determine how to process the bytes received. The IEEE Std 1003.1-200x does not specify whether the setting of O_NONBLOCK takes precedence over MIN or TIME settings. Therefore, if O_NONBLOCK is set, *read*( ) may return immediately, regardless of the setting of MIN or TIME. Also, if no data is available, *read*( ) may either return 0, or return −1 with *errno* set to [EAGAIN].

MIN represents the minimum number of bytes that should be received when the *read*( ) function returns successfully. TIME is a timer of 0.1 second granularity that is used to time out bursty and short-term data transmissions. If MIN is greater than {MAX_INPUT}, the response to the request is undefined. The four possible values for MIN and TIME and their interactions are described below.

Case A: MIN>0, TIME>0

In case A, TIME serves as an inter-byte timer which shall be activated after the first byte is received. Since it is an inter-byte timer, it shall be reset after a byte is received. The interaction between MIN and TIME is as follows. As soon as one byte is received, the inter-byte timer shall be started. If MIN bytes are received before the inter-byte timer expires (remember that the timer is reset upon receipt of each byte), the read shall be satisfied. If the timer expires before MIN bytes are received, the characters received to that point shall be returned to the user. Note that if TIME expires at least one byte shall be returned because the timer would not have been enabled unless a byte was received. In this case (MIN>0, TIME>0) the read shall block until the MIN and TIME mechanisms are activated by the receipt of the first byte, or a signal is received. If data is in the buffer at the time of the *read*( ), the result shall be as if data has been received immediately after the *read*( ).

Case B: MIN>0, TIME=0

In case B, since the value of TIME is zero, the timer plays no role and only MIN is significant. A pending read shall not be satisfied until MIN bytes are received (that is, the pending read shall block until MIN bytes are received), or a signal is received. A program that uses case B to read record-based terminal I/O may block indefinitely in the read operation.

Case C: MIN=0, TIME>0

In case C, since MIN=0, TIME no longer represents an inter-byte timer. It now serves as a read timer that shall be activated as soon as the *read*( ) function is processed. A read shall be satisfied as soon as a single byte is received or the read timer expires. Note that in case C if the timer expires, no bytes shall be returned. If the timer does not expire, the only way the read can be satisfied is if a byte is received. If bytes are not received, the read shall not block indefinitely waiting for a byte; if no byte is received within TIME*0.1 seconds after the read is initiated, the *read*( ) shall return a value of zero, having read no data. If data is in the buffer at the time of the *read*( ), the timer shall be started as if data has been received immediately after the *read*( ).

Case D: MIN=0, TIME=0

The minimum of either the number of bytes requested or the number of bytes currently available shall be returned without waiting for more bytes to be input. If no characters are available, *read*( ) shall return a value of zero, having read no data.

11.1.8 Writing Data and Output Processing

When a process writes one or more bytes to a terminal device file, they are processed according to the **c_oflag** field (see Section 11.2.3 (on page 190)). The implementation may provide a buffering mechanism; as such, when a call to *write*( ) completes, all of the bytes written have been scheduled for transmission to the device, but the transmission has not necessarily completed. See *write*( ) for the effects of O_NONBLOCK on *write*( ).

11.1.9 Special Characters

Certain characters have special functions on input or output or both. These functions are summarized as follows:

INTR	Special character on input, which is recognized if the ISIG flag is set. Generates a SIGINT signal which is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the INTR character shall be discarded when processed.
QUIT	Special character on input, which is recognized if the ISIG flag is set. Generates a SIGQUIT signal which is sent to all processes in the foreground process group for which the terminal is the controlling terminal. If ISIG is set, the QUIT character shall be discarded when processed.
ERASE	Special character on input, which is recognized if the ICANON flag is set. Erases the last character in the current line; see Section 11.1.6 (on page 185). It shall not erase beyond the start of a line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the ERASE character shall be discarded when processed.
KILL	Special character on input, which is recognized if the ICANON flag is set. Deletes the entire line, as delimited by an NL, EOF, or EOL character. If ICANON is set, the KILL character shall be discarded when processed.
EOF	Special character on input, which is recognized if the ICANON flag is set. When received, all the bytes waiting to be read are immediately passed to the process without waiting for a newline, and the EOF is discarded. Thus, if there are no bytes waiting (that is, the EOF occurred at the beginning of a line), a byte count of zero shall be returned from the *read*( ), representing an end-of-file indication. If ICANON is set, the EOF character shall be discarded when processed.
NL	Special character on input, which is recognized if the ICANON flag is set. It is the line delimiter newline. It cannot be changed.
EOL	Special character on input, which is recognized if the ICANON flag is set. It is an additional line delimiter, like NL.
SUSP	If the ISIG flag is set, receipt of the SUSP character shall cause a SIGTSTP signal to be sent to all processes in the foreground process group for which the terminal is the controlling terminal, and the SUSP character shall be discarded when processed.
STOP	Special character on both input and output, which is recognized if the IXON (output control) or IXOFF (input control) flag is set. Can be used to suspend output temporarily. It is useful with CRT terminals to prevent output from disappearing

before it can be read. If IXON is set, the STOP character shall be discarded when processed.

START Special character on both input and output, which is recognized if the IXON (output control) or IXOFF (input control) flag is set. Can be used to resume output that has been suspended by a STOP character. If IXON is set, the START character shall be discarded when processed.

CR Special character on input, which is recognized if the ICANON flag is set; it is the carriage-return character. When ICANON and ICRNL are set and IGNCR is not set, this character shall be translated into an NL, and shall have the same effect as an NL character.

The NL and CR characters cannot be changed. It is implementation-defined whether the START and STOP characters can be changed. The values for INTR, QUIT, ERASE, KILL, EOF, EOL, and SUSP shall be changeable to suit individual tastes. Special character functions associated with changeable special control characters can be disabled individually.

If two or more special characters have the same value, the function performed when that character is received is undefined.

A special character is recognized not only by its value, but also by its context; for example, an implementation may support multi-byte sequences that have a meaning different from the meaning of the bytes when considered individually. Implementations may also support additional single-byte functions. These implementation-defined multi-byte or single-byte functions shall be recognized only if the IEXTEN flag is set; otherwise, data is received without interpretation, except as required to recognize the special characters defined in this section.

XSI If IEXTEN is set, the ERASE, KILL, and EOF characters can be escaped by a preceding '\' character, in which case no special function shall occur.

11.1.10 Modem Disconnect

If a modem disconnect is detected by the terminal interface for a controlling terminal, and if CLOCAL is not set in the **c_cflag** field for the terminal (see Section 11.2.4 (on page 192)), the SIGHUP signal shall be sent to the controlling process for which the terminal is the controlling terminal. Unless other arrangements have been made, this shall cause the controlling process to terminate (see *exit*()). Any subsequent read from the terminal device shall return the value of zero, indicating end-of-file; see *read*(). Thus, processes that read a terminal file and test for end-of-file can terminate appropriately after a disconnect. If the EIO condition as specified in *read*() also exists, it is unspecified whether on EOF condition or the [EIO] is returned. Any subsequent *write*() to the terminal device shall return −1, with *errno* set to [EIO], until the device is closed.

11.1.11 Closing a Terminal Device File

The last process to close a terminal device file shall cause any output to be sent to the device and any input to be discarded. If HUPCL is set in the control modes and the communications port supports a disconnect function, the terminal device shall perform a disconnect.

11.2 Parameters that Can be Set

11.2.1 The termios Structure

Routines that need to control certain terminal I/O characteristics shall do so by using the **termios** structure as defined in the **<termios.h>** header. The members of this structure include (but are not limited to):

Member Type	Array Size	Member Name	Description
tcflag_t		**c_iflag**	Input modes.
tcflag_t		**c_oflag**	Output modes.
tcflag_t		**c_cflag**	Control modes.
tcflag_t		**c_lflag**	Local modes.
cc_t	NCCS	**c_cc**[ ]	Control characters.

The types **tcflag_t** and **cc_t** are defined in the **<termios.h>** header. They shall be unsigned integer types.

11.2.2 Input Modes

Values of the **c_iflag** field describe the basic terminal input control, and are composed of the bitwise-inclusive OR of the masks shown, which shall be bitwise-distinct. The mask name symbols in this table are defined in **<termios.h>**: |

Mask Name	Description
BRKINT	Signal interrupt on break.
ICRNL	Map CR to NL on input.
IGNBRK	Ignore break condition.
IGNCR	Ignore CR.
IGNPAR	Ignore characters with parity errors.
INLCR	Map NL to CR on input.
INPCK	Enable input parity check.
ISTRIP	Strip character.
XSI IXANY	Enable any character to restart output.
IXOFF	Enable start/stop input control.
IXON	Enable start/stop output control.
PARMRK	Mark parity errors.

In the context of asynchronous serial data transmission, a break condition shall be defined as a sequence of zero-valued bits that continues for more than the time to send one byte. The entire sequence of zero-valued bits is interpreted as a single break condition, even if it continues for a time equivalent to more than one byte. In contexts other than asynchronous serial data transmission, the definition of a break condition is implementation-defined.

If IGNBRK is set, a break condition detected on input shall be ignored; that is, not put on the input queue and therefore not read by any process. If IGNBRK is not set and BRKINT is set, the break condition shall flush the input and output queues, and if the terminal is the controlling terminal of a foreground process group, the break condition shall generate a single SIGINT signal to that foreground process group. If neither IGNBRK nor BRKINT is set, a break condition shall be read as a single 0x00, or if PARMRK is set, as 0xff 0x00 0x00.

If IGNPAR is set, a byte with a framing or parity error (other than break) shall be ignored.

If PARMRK is set, and IGNPAR is not set, a byte with a framing or parity error (other than break) shall be given to the application as the three-byte sequence 0xff 0x00 X, where 0xff 0x00 is a two-byte flag preceding each sequence and X is the data of the byte received in error. To avoid ambiguity in this case, if ISTRIP is not set, a valid byte of 0xff is given to the application as 0xff 0xff. If neither PARMRK nor IGNPAR is set, a framing or parity error (other than break) shall be given to the application as a single byte 0x00.

If INPCK is set, input parity checking shall be enabled. If INPCK is not set, input parity checking shall be disabled, allowing output parity generation without input parity errors. Note that whether input parity checking is enabled or disabled is independent of whether parity detection is enabled or disabled (see Section 11.2.4 (on page 192)). If parity detection is enabled but input parity checking is disabled, the hardware to which the terminal is connected shall recognize the parity bit, but the terminal special file shall not check whether or not this bit is correctly set.

If ISTRIP is set, valid input bytes shall first be stripped to seven bits; otherwise, all eight bits shall be processed.

If INLCR is set, a received NL character shall be translated into a CR character. If IGNCR is set, a received CR character shall be ignored (not read). If IGNCR is not set and ICRNL is set, a received CR character shall be translated into an NL character.

XSI If IXANY is set, any input character shall restart output that has been suspended.

If IXON is set, start/stop output control shall be enabled. A received STOP character shall suspend output and a received START character shall restart output. When IXON is set, START and STOP characters are not read, but merely perform flow control functions. When IXON is not set, the START and STOP characters shall be read.

If IXOFF is set, start/stop input control shall be enabled. The system shall transmit STOP characters, which are intended to cause the terminal device to stop transmitting data, as needed to prevent the input queue from overflowing and causing implementation-defined behavior, and shall transmit START characters, which are intended to cause the terminal device to resume transmitting data, as soon as the device can continue transmitting data without risk of overflowing the input queue. The precise conditions under which STOP and START characters are transmitted are implementation-defined.

The initial input control value after *open*( ) is implementation-defined.

11.2.3 Output Modes

The **c_oflag** field specifies the terminal interface's treatment of output, and is composed of the bitwise-inclusive OR of the masks shown, which shall be bitwise-distinct. The mask name symbols in this table are defined in **<termios.h>**:

Mask Name	Description
OPOST	Perform output processing.
ONLCR	Map NL to CR-NL on output.
OCRNL	Map CR to NL on output.
ONOCR	No CR output at column 0.
ONLRET	NL performs CR function.
OFILL	Use fill characters for delay.
OFDEL	Fill is DEL, else NUL.
NLDLY	Select newline delays:
NL0	Newline character type 0.
NL1	Newline character type 1.
CRDLY	Select carriage-return delays:
CR0	Carriage-return delay type 0.
CR1	Carriage-return delay type 1.
CR2	Carriage-return delay type 2.
CR3	Carriage-return delay type 3.
TABDLY	Select horizontal-tab delays:
TAB0	Horizontal-tab delay type 0.
TAB1	Horizontal-tab delay type 1.
TAB2	Horizontal-tab delay type 2.
TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:
BS0	Backspace-delay type 0.
BS1	Backspace-delay type 1.
VTDLY	Select vertical-tab delays:
VT0	Vertical-tab delay type 0.
VT1	Vertical-tab delay type 1.
FFDLY	Select form-feed delays:
FF0	Form-feed delay type 0.
FF1	Form-feed delay type 1.

If OPOST is set, output data shall be post-processed as described below, so that lines of text are modified to appear appropriately on the terminal device; otherwise, characters shall be transmitted without change.

XSI If ONLCR is set, the NL character shall be transmitted as the CR-NL character pair. If OCRNL is set, the CR character shall be transmitted as the NL character. If ONOCR is set, no CR character shall be transmitted when at column 0 (first position). If ONLRET is set, the NL character is assumed to do the carriage-return function; the column pointer shall be set to 0 and the delays specified for CR shall be used. Otherwise, the NL character is assumed to do just the line-feed function; the column pointer remains unchanged. The column pointer shall also be set to 0 if the CR character is actually transmitted.

The delay bits specify how long transmission stops to allow for mechanical or other movement when certain characters are sent to the terminal. In all cases a value of 0 shall indicate no delay. If OFILL is set, fill characters shall be transmitted for delay instead of a timed delay. This is useful for high baud rate terminals which need only a minimal delay. If OFDEL is set, the fill character shall be DEL; otherwise, NUL.

If a form-feed or vertical-tab delay is specified, it shall last for about 2 seconds.

New-line delay shall last about 0.10 seconds. If ONLRET is set, the carriage-return delays shall be used instead of the newline delays. If OFILL is set, two fill characters shall be transmitted.

Carriage-return delay type 1 shall be dependent on the current column position, type 2 shall be about 0.10 seconds, and type 3 shall be about 0.15 seconds. If OFILL is set, delay type 1 shall transmit two fill characters, and type 2, four fill characters.

Horizontal-tab delay type 1 shall be dependent on the current column position. Type 2 shall be about 0.10 seconds. Type 3 specifies that tabs shall be expanded into spaces. If OFILL is set, two fill characters shall be transmitted for any delay.

Backspace delay shall last about 0.05 seconds. If OFILL is set, one fill character shall be transmitted.

The actual delays depend on line speed and system load.

The initial output control value after *open*( ) is implementation-defined.

11.2.4 Control Modes

The **c_cflag** field describes the hardware control of the terminal, and is composed of the bitwise-inclusive OR of the masks shown, which shall be bitwise-distinct. The mask name symbols in this table are defined in **<termios.h>**; not all values specified are required to be supported by the underlying hardware:

Mask Name	Description
CLOCAL	Ignore modem status lines.
CREAD	Enable receiver.
CSIZE	Number of bits transmitted or received per byte:
CS5	5 bits
CS6	6 bits
CS7	7 bits
CS8	8 bits.
CSTOPB	Send two stop bits, else one.
HUPCL	Hang up on last close.
PARENB	Parity enable.
PARODD	Odd parity, else even.

In addition, the input and output baud rates are stored in the **termios** structure. The symbols in the following table are defined in **<termios.h>**. Not all values specified are required to be supported by the underlying hardware.

Name	Description	Name	Description
B0	Hang up	B600	600 baud
B50	50 baud	B1200	1200 baud
B75	75 baud	B1800	1800 baud
B110	110 baud	B2400	2400 baud
B134	134.5 baud	B4800	4800 baud
B150	150 baud	B9600	9600 baud
B200	200 baud	B19200	19200 baud
B300	300 baud	B38400	38400 baud

The following functions are provided for getting and setting the values of the input and output baud rates in the **termios** structure: *cfgetispeed*( ), *cfgetospeed*( ), *cfsetispeed*( ), and *cfsetospeed*( ). The effects on the terminal device shall not become effective and not all errors need be detected until the *tcsetattr*( ) function is successfully called.

The CSIZE bits shall specify the number of transmitted or received bits per byte. If ISTRIP is not set, the value of all the other bits is unspecified. If ISTRIP is set, the value of all but the 7 low-order bits shall be zero, but the value of any other bits beyond CSIZE is unspecified when read. CSIZE shall not include the parity bit, if any. If CSTOPB is set, two stop bits shall be used; otherwise, one stop bit. For example, at 110 baud, two stop bits are normally used.

If CREAD is set, the receiver shall be enabled; otherwise, no characters shall be received.

If PARENB is set, parity generation and detection shall be enabled and a parity bit is added to each byte. If parity is enabled, PARODD shall specify odd parity if set; otherwise, even parity shall be used.

If HUPCL is set, the modem control lines for the port shall be lowered when the last process with the port open closes the port or the process terminates. The modem connection shall be broken.

If CLOCAL is set, a connection shall not depend on the state of the modem status lines. If CLOCAL is clear, the modem status lines shall be monitored.

Under normal circumstances, a call to the *open*( ) function shall wait for the modem connection to complete. However, if the O_NONBLOCK flag is set (see *open*( )) or if CLOCAL has been set, the *open*( ) function shall return immediately without waiting for the connection.

If the object for which the control modes are set is not an asynchronous serial connection, some of the modes may be ignored; for example, if an attempt is made to set the baud rate on a network connection to a terminal on another host, the baud rate need not be set on the connection between that terminal and the machine to which it is directly connected.

The initial hardware control value after *open*( ) is implementation-defined.

11.2.5 Local Modes

The **c_lflag** field of the argument structure is used to control various functions. It is composed of the bitwise-inclusive OR of the masks shown, which shall be bitwise-distinct. The mask name symbols in this table are defined in **<termios.h>**; not all values specified are required to be supported by the underlying hardware:

Mask Name	Description
ECHO	Enable echo.
ECHOE	Echo ERASE as an error correcting backspace.
ECHOK	Echo KILL.
ECHONL	Echo <newline>.
ICANON	Canonical input (erase and kill processing).
IEXTEN	Enable extended (implementation-defined) functions.
ISIG	Enable signals.
NOFLSH	Disable flush after interrupt, quit or suspend.
TOSTOP	Send SIGTTOU for background output.

If ECHO is set, input characters shall be echoed back to the terminal. If ECHO is clear, input characters shall not be echoed.

If ECHOE and ICANON are set, the ERASE character shall cause the terminal to erase, if possible, the last character in the current line from the display. If there is no character to erase, an implementation may echo an indication that this was the case, or do nothing.

If ECHOK and ICANON are set, the KILL character shall either cause the terminal to erase the line from the display or shall echo the newline character after the KILL character.

If ECHONL and ICANON are set, the newline character shall be echoed even if ECHO is not set.

If ICANON is set, canonical processing shall be enabled. This enables the erase and kill edit functions, and the assembly of input characters into lines delimited by NL, EOF, and EOL, as described in Section 11.1.6 (on page 185).

If ICANON is not set, read requests shall be satisfied directly from the input queue. A read shall not be satisfied until at least MIN bytes have been received or the timeout value TIME expired between bytes. The time value represents tenths of a second. See Section 11.1.7 (on page 186) for more details.

If IEXTEN is set, implementation-defined functions shall be recognized from the input data. It is implementation-defined how IEXTEN being set interacts with ICANON, ISIG, IXON, or IXOFF. If IEXTEN is not set, implementation-defined functions shall not be recognized and the corresponding input characters are processed as described for ICANON, ISIG, IXON, and IXOFF.

If ISIG is set, each input character shall be checked against the special control characters INTR, QUIT, and SUSP. If an input character matches one of these control characters, the function associated with that character shall be performed. If ISIG is not set, no checking shall be done. Thus these special input functions are possible only if ISIG is set.

If NOFLSH is set, the normal flush of the input and output queues associated with the INTR, QUIT, and SUSP characters shall not be done.

If TOSTOP is set, the signal SIGTTOU shall be sent to the process group of a process that tries to write to its controlling terminal if it is not in the foreground process group for that terminal. This signal, by default, stops the members of the process group. Otherwise, the output generated by that process shall be output to the current output stream. Processes that are blocking or ignoring SIGTTOU signals are excepted and allowed to produce output, and the SIGTTOU signal shall not be sent.

The initial local control value after *open*( ) is implementation-defined.

11.2.6 Special Control Characters

The special control character values shall be defined by the array **c_cc**. The subscript name and description for each element in both canonical and non-canonical modes are as follows:

Subscript Usage		
Canonical Mode	**Non-Canonical Mode**	**Description**
VEOF		EOF character
VEOL		EOL character
VERASE		ERASE character
VINTR	VINTR	INTR character
VKILL		KILL character
	VMIN	MIN value
VQUIT	VQUIT	QUIT character
VSUSP	VSUSP	SUSP character
	VTIME	TIME value
VSTART	VSTART	START character
VSTOP	VSTOP	STOP character

The subscript values are unique, except that the VMIN and VTIME subscripts may have the same values as the VEOF and VEOL subscripts, respectively.

Implementations that do not support changing the START and STOP characters may ignore the character values in the **c_cc** array indexed by the VSTART and VSTOP subscripts when *tcsetattr*( ) is called, but shall return the value in use when *tcgetattr*( ) is called.

The initial values of all control characters are implementation-defined.

If the value of one of the changeable special control characters (see Section 11.1.9 (on page 187)) is _POSIX_VDISABLE, that function shall be disabled; that is, no input data is recognized as the disabled special character. If ICANON is not set, the value of _POSIX_VDISABLE has no special meaning for the VMIN and VTIME entries of the **c_cc** array.

Chapter 12

Utility Conventions

12.1 Utility Argument Syntax

This section describes the argument syntax of the standard utilities and introduces terminology used throughout IEEE Std 1003.1-200x for describing the arguments processed by the utilities.

Within IEEE Std 1003.1-200x, a special notation is used for describing the syntax of a utility's arguments. Unless otherwise noted, all utility descriptions use this notation, which is illustrated by this example (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1, Simple Commands):

```
utility_name[−a][−b][−c option_argument]
    [−d|−e][−foption_argument][operand...]
```

The notation used for the SYNOPSIS sections imposes requirements on the implementors of the standard utilities and provides a simple reference for the application developer or system user.

1. The utility in the example is named *utility_name*. It is followed by *options*, *option-arguments*, and *operands*. The arguments that consist of hyphens and single letters or digits, such as `'a'`, are known as *options* (or, historically, *flags*). Certain options are followed by an *option-argument*, as shown with [−**c** *option_argument*]. The arguments following the last options and option-arguments are named *operands*.

2. Option-arguments are sometimes shown separated from their options by <blank>s, sometimes directly adjacent. This reflects the situation that in some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. The Utility Syntax Guidelines in Section 12.2 (on page 199) require that the option be a separate argument from its option-argument, but there are some exceptions in IEEE Std 1003.1-200x to ensure continued operation of historical applications:

   a. If the SYNOPSIS of a standard utility shows a space character between an option and option-argument (as with [−**c** *option_argument*] in the example), a conforming application shall use separate arguments for that option and its option-argument.

   b. If a space character is not shown (as with [−**f***option_argument*] in the example), a conforming application shall place an option and its option-argument directly adjacent in the same argument string, without intervening <blank>s.

   c. Notwithstanding the preceding requirements on conforming applications, a conforming system shall permit, but shall not require, an application to specify options and option-arguments as separate arguments whether or not a space character is shown on the synopsis line, XSI except in those cases (marked with the XSI portability warning) where an option-argument is optional and no separation can be used.

   d. A standard utility may also be implemented to operate correctly when the required separation into multiple arguments is violated by a non-conforming application.

   In summary, the following table shows allowable combinations:

	SYNOPSIS Shows:		
	−a *arg*	−b*arg*	−c[*arg*]
Conforming application shall use:	−a *arg*	−b*arg*	N/A
System shall support:	−a *arg*	−b*arg*	−c*arg* or −c
System may support:	−a*arg*	−b *arg*	

3. Options are usually listed in alphabetical order unless this would make the utility description more confusing. There are no implied relationships between the options based upon the order in which they appear, unless otherwise stated in the OPTIONS section, or unless the exception in Guideline 11 of Section 12.2 (on page 199) applies. If an option that does not have option-arguments is repeated, the results are undefined, unless otherwise stated.

4. Frequently, names of parameters that require substitution by actual values are shown with embedded underscores. Alternatively, parameters are shown as follows:

   <*parameter name*>

   The angle brackets are used for the symbolic grouping of a phrase representing a single parameter and conforming applications shall not include them in data submitted to the utility.

5. When a utility has only a few permissible options, they are sometimes shown individually, as in the example. Utilities with many flags generally show all of the individual flags (that do not take option-arguments) grouped, as in:

   utility_name **[**−abcDxyz**][**−p *arg***][***operand***]**

   Utilities with very complex arguments may be shown as follows:

   utility_name **[**options**][***operands***]**

6. Unless otherwise specified, whenever an operand or option-argument is, or contains, a numeric value:

   - The number is interpreted as a decimal integer.
   - Numerals in the range 0 to 2 147 483 647 are syntactically recognized as numeric values.
   - When the utility description states that it accepts negative numbers as operands or option-arguments, numerals in the range −2 147 483 647 to 2 147 483 647 are syntactically recognized as numeric values.
   - Ranges greater than those listed here are allowed.

   This does not mean that all numbers within the allowable range are necessarily semantically correct. A standard utility that accepts an option-argument or operand that is to be interpreted as a number, and for which a range of values smaller than that shown above is permitted by the IEEE Std 1003.1-200x, describes that smaller range along with the description of the option-argument or operand. If an error is generated, the utility's diagnostic message shall indicate that the value is out of the supported range, not that it is syntactically incorrect.

7. Arguments or option-arguments enclosed in the '[' and ']' notation are optional and can be omitted. Conforming applications shall not include the '[' and ']' symbols in data submitted to the utility.

8. Arguments separated by the '|' vertical bar notation are mutually-exclusive. Conforming applications shall not include the '|' symbol in data submitted to the utility.

Alternatively, mutually-exclusive options and operands may be listed with multiple synopsis lines. For example:

```
utility_name -d[-a][-c option_argument][operand...]
utility_name[-a][-b][operand...]
```

When multiple synopsis lines are given for a utility, it is an indication that the utility has mutually-exclusive arguments. These mutually-exclusive arguments alter the functionality of the utility so that only certain other arguments are valid in combination with one of the mutually-exclusive arguments. Only one of the mutually-exclusive arguments is allowed for invocation of the utility. Unless otherwise stated in an accompanying OPTIONS section, the relationships between arguments depicted in the SYNOPSIS sections are mandatory requirements placed on conforming applications. The use of conflicting mutually-exclusive arguments produces undefined results, unless a utility description specifies otherwise. When an option is shown without the '[' and ']' brackets, it means that option is required for that version of the SYNOPSIS. However, it is not required to be the first argument, as shown in the example above, unless otherwise stated.

9. Ellipses (`"..."`) are used to denote that one or more occurrences of an option or operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands can be specified. The forms:

```
utility_name -f option_argument...[operand...]
utility_name [-g option_argument]...[operand...]
```

   indicate that multiple occurrences of the option and its option-argument preceding the ellipses are valid, with semantics as indicated in the OPTIONS section of the utility. (See also Guideline 11 in Section 12.2.) In the first example, each option-argument requires a preceding **−f** and at least one **−f** *option_argument* must be given.

10. When the synopsis line is too long to be printed on a single line in the Shell and Utilities volume of IEEE Std 1003.1-200x, the indented lines following the initial line are continuation lines. An actual use of the command would appear on a single logical line.

12.2 Utility Syntax Guidelines

The following guidelines are established for the naming of utilities and for the specification of options, option-arguments, and operands. The *getopt*( ) function in the System Interfaces volume of IEEE Std 1003.1-200x assists utilities in handling options and operands that conform to these guidelines.

Operands and option-arguments can contain characters not specified in the portable character set.

The guidelines are intended to provide guidance to the authors of future utilities, such as those written specific to a local system or that are components of a larger application. Some of the standard utilities do not conform to all of these guidelines; in those cases, the OPTIONS sections describe the deviations.

Guideline 1: Utility names should be between two and nine characters, inclusive.

Guideline 2: Utility names should include lowercase letters (the **lower** character classification) and digits only from the portable character set.

Guideline 3: Each option name should be a single alphanumeric character (the **alnum** character classification) from the portable character set. The **−W** (capital-W) option shall be reserved for vendor options.

Multi-digit options should not be allowed.

Guideline 4: All options should be preceded by the ’−’ delimiter character.

Guideline 5: Options without option-arguments should be accepted when grouped behind one ’−’ delimiter.

Guideline 6: Each option and option-argument should be a separate argument, except as noted in Section 12.1 (on page 197), item (2).

Guideline 7: Option-arguments should not be optional.

Guideline 8: When multiple option-arguments are specified to follow a single option, they should be presented as a single argument, using commas within that argument or <blank>s within that argument to separate them.

Guideline 9: All options should precede operands on the command line.

Guideline 10: The argument −− should be accepted as a delimiter indicating the end of options. Any following arguments should be treated as operands, even if they begin with the ’−’ character. The −− argument should not be used as an option or as an operand.

Guideline 11: The order of different options relative to one another should not matter, unless the options are documented as mutually-exclusive and such an option is documented to override any incompatible options preceding it. If an option that has option-arguments is repeated, the option and option-argument combinations should be interpreted in the order specified on the command line.

Guideline 12: The order of operands may matter and position-related interpretations should be determined on a utility-specific basis.

Guideline 13: For utilities that use operands to represent files to be opened for either reading or writing, the ’−’ operand should be used only to mean standard input (or standard output when it is clear from context that an output file is being specified).

The utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x that claim conformance to these guidelines shall conform completely to these guidelines as if these guidelines contained the term ‘‘shall’’ instead of ‘‘should’’. On some implementations, the utilities accept usage in violation of these guidelines for backward compatibility as well as accepting the required form.

It is recommended that all future utilities and applications use these guidelines to enhance user portability. The fact that some historical utilities could not be changed (to avoid breaking existing applications) should not deter this future goal.

Chapter 13

Headers

This chapter describes the contents of headers.

Headers contain function prototypes, the definition of symbolic constants, common structures, preprocessor macros, and defined types. Each function in the System Interfaces volume of IEEE Std 1003.1-200x specifies the headers that an application shall include in order to use that function. In most cases, only one header is required. These headers are present on an application development system; they need not be present on the target execution system.

13.1 Format of Entries

The entries in this chapter are based on a common format as follows. The only sections relating to conformance are the SYNOPSIS and DESCRIPTION.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described.

DESCRIPTION

This section describes the functionality of the header.

APPLICATION USAGE

This section is non-normative.

This section gives warnings and advice to application writers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

RATIONALE

This section is non-normative.

This section contains historical information concerning the contents of this volume of IEEE Std 1003.1-200x and why features were included or discarded by the standard developers.

FUTURE DIRECTIONS

This section is non-normative.

This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

This section is non-normative.

This section gives references to related information.

CHANGE HISTORY

This section is non-normative.

This section shows the derivation of the entry and any significant changes that have been made to it.

NAME

aio.h — asynchronous input and output (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>
```

DESCRIPTION

The **<aio.h>** header shall define the **aiocb** structure which shall include at least the following members:

`int`	`aio_fildes`	File descriptor.
`off_t`	`aio_offset`	File offset.
`volatile void`	`*aio_buf`	Location of buffer.
`size_t`	`aio_nbytes`	Length of transfer.
`int`	`aio_reqprio`	Request priority offset.
`struct sigevent`	`aio_sigevent`	Signal number and value.
`int`	`aio_lio_opcode`	Operation to be performed.

This header shall also include the following constants:

AIO_CANCELED	A return value indicating that all requested operations have been canceled.
AIO_NOTCANCELED	A return value indicating that some of the requested operations could not be canceled since they are in progress.
AIO_ALLDONE	A return value indicating that none of the requested operations could be canceled since they are already complete.
LIO_WAIT	A *lio_listio*( ) synchronization operation indicating that the calling thread is to suspend until the *lio_listio*( ) operation is complete.
LIO_NOWAIT	A *lio_listio*( ) synchronization operation indicating that the calling thread is to continue execution while the *lio_listio*( ) operation is being performed, and no notification is given when the operation is complete.
LIO_READ	A *lio_listio*( ) element operation option requesting a read.
LIO_WRITE	A *lio_listio*( ) element operation option requesting a write.
LIO_NOP	A *lio_listio*( ) element operation option indicating that no transfer is requested.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int      aio_cancel(int, struct aiocb *);
int      aio_error(const struct aiocb *);
int      aio_fsync(int, struct aiocb *);
int      aio_read(struct aiocb *);
ssize_t  aio_return(struct aiocb *);
int      aio_suspend(const struct aiocb *const[], int,
             const struct timespec *);
int      aio_write(struct aiocb *);
int      lio_listio(int, struct aiocb *restrict const[restrict], int,
             struct sigevent *restrict);
```

Inclusion of the **<aio.h>** header may make visible symbols defined in the headers **<fcntl.h>**, **<signal.h>**, **<sys/types.h>**, and **<time.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<fcntl.h>, **<signal.h>**, **<sys/types.h>**, **<time.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *fsync*( ), *lseek*( ), *read*( ), *write*( )

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The **<aio.h>** header is marked as part of the Asynchronous Input and Output option.

The description of the constants is expanded.

The **restrict** keyword is added to the prototype for *lio_listio*( ).

NAME

arpa/inet.h — definitions for internet operations

SYNOPSIS

```
#include <arpa/inet.h>
```

DESCRIPTION

The **in_port_t** and **in_addr_t** types shall be defined as described in **<netinet/in.h>**.

The **in_addr** structure shall be defined as described in **<netinet/in.h>**.

IP6 The INET_ADDRSTRLEN and INET6_ADDRSTRLEN macros shall be defined as described in **<netinet/in.h>**.

The following shall either be declared as functions, defined as macros, or both. If functions are declared, function prototypes shall be provided.

```
uint32_t htonl(uint32_t);
uint16_t htons(uint16_t);
uint32_t ntohl(uint32_t);
uint16_t ntohs(uint16_t);
```

The **uint32_t** and **uint16_t** types shall be defined as described in **<inttypes.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
in_addr_t    inet_addr(const char *);
char        *inet_ntoa(struct in_addr);
const char  *inet_ntop(int, const void *restrict, char *restrict,
                 socklen_t);
int          inet_pton(int, const char *restrict, void *restrict);
```

Inclusion of the **<arpa/inet.h>** header may also make visible all symbols from **<netinet/in.h>** and **<inttypes.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<netinet/in.h>, **<inttypes.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *htonl*( ), *inet_addr*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the prototypes for *inet_ntop*( ) and *inet_pton*( ).

NAME

assert.h — verify program assertion

SYNOPSIS

```
#include <assert.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<assert.h>** header shall define the *assert*( ) macro. It refers to the macro NDEBUG which is not defined in the header. If NDEBUG is defined as a macro name before the inclusion of this header, the *assert*( ) macro shall be defined simply as:

```
#define assert(ignore)((void) 0)
```

Otherwise, the macro behaves as described in *assert*( ).

The *assert*( ) macro shall be redefined according to the current state of NDEBUG each time **<assert.h>** is included.

The *assert*( ) macro shall be implemented as a macro, not as a function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *assert*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The definition of the *assert*( ) macro is changed for alignment with the ISO/IEC 9899:1999 standard.

NAME

complex.h — complex arithmetic

SYNOPSIS

```
#include <complex.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<complex.h>** header shall define the following macros:

complex	Expands to **_Complex**.
_Complex_I	Expands to a constant expression of type **const float _Complex**, with the value of the imaginary unit (that is, a number such that $i^2=-1$).
imaginary	Expands to **_Imaginary**.
_Imaginary_I	Expands to a constant expression of type **const float _Imaginary** with the value of the imaginary unit.
I	Expands to either _Imaginary_I or _Complex_I. If _Imaginary_I is not defined, I expands to _Complex_I.

The macros imaginary and _Imaginary_I shall be defined if and only if the implementation supports imaginary types.

An application may undefine and then, perhaps, redefine the complex, imaginary, and I macros.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
double               cabs(double complex);
float                cabsf(float complex);
long double          cabsl(long double complex);
double complex       cacos(double complex);
float complex        cacosf(float complex);
double complex       cacosh(double complex);
float complex        cacoshf(float complex);
long double complex  cacoshl(long double complex);
long double complex  cacosl(long double complex);
double               carg(double complex);
float                cargf(float complex);
long double          cargl(long double complex);
double complex       casin(double complex);
float complex        casinf(float complex);
double complex       casinh(double complex);
float complex        casinhf(float complex);
long double complex  casinhl(long double complex);
long double complex  casinl(long double complex);
double complex       catan(double complex);
float complex        catanf(float complex);
double complex       catanh(double complex);
float complex        catanhf(float complex);
long double complex  catanhl(long double complex);
long double complex  catanl(long double complex);
```

```
double complex       ccos(double complex);
float complex        ccosf(float complex);
double complex       ccosh(double complex);
float complex        ccoshf(float complex);
long double complex  ccoshl(long double complex);
long double complex  ccosl(long double complex);
double complex       cexp(double complex);
float complex        cexpf(float complex);
long double complex  cexpl(long double complex);
double               cimag(double complex);
float                cimagf(float complex);
long double          cimagl(long double complex);
double complex       clog(double complex);
float complex        clogf(float complex);
long double complex  clogl(long double complex);
double complex       conj(double complex);
float complex        conjf(float complex);
long double complex  conjl(long double complex);
double complex       cpow(double complex, double complex);
float complex        cpowf(float complex, float complex);
long double complex  cpowl(long double complex, long double complex);
double complex       cproj(double complex);
float complex        cprojf(float complex);
long double complex  cprojl(long double complex);
double               creal(double complex);
float                crealf(float complex);
long double          creall(long double complex);
double complex       csin(double complex);
float complex        csinf(float complex);
double complex       csinh(double complex);
float complex        csinhf(float complex);
long double complex  csinhl(long double complex);
long double complex  csinl(long double complex);
double complex       csqrt(double complex);
float complex        csqrtf(float complex);
long double complex  csqrtl(long double complex);
double complex       ctan(double complex);
float complex        ctanf(float complex);
double complex       ctanh(double complex);
float complex        ctanhf(float complex);
long double complex  ctanhl(long double complex);
long double complex  ctanl(long double complex);
```

APPLICATION USAGE

Values are interpreted as radians, not degrees.

RATIONALE

The choice of *I* instead of *i* for the imaginary unit concedes to the widespread use of the identifier *i* for other purposes. The application can use a different identifier, say *j*, for the imaginary unit by following the inclusion of the **<complex.h>** header with:

```
#undef I
#define j _Imaginary_I
```

An *I* suffix to designate imaginary constants is not required, as multiplication by *I* provides a sufficiently convenient and more generally useful notation for imaginary terms. The corresponding real type for the imaginary unit is **float**, so that use of *I* for algorithmic or notational convenience will not result in widening types.

On systems with imaginary types, the application has the ability to control whether use of the macro I introduces an imaginary type, by explicitly defining I to be _Imaginary_I or _Complex_I. Disallowing imaginary types is useful for some applications intended to run on implementations without support for such types.

The macro _Imaginary_I provides a test for whether imaginary types are supported.

The *cis*( ) function (*cos*(*x*) + *I***sin*(*x*)) was considered but rejected because its implementation is easy and straightforward, even though some implementations could compute sine and cosine more efficiently in tandem.

FUTURE DIRECTIONS

The following function names and the same names suffixed with *f* or *l* are reserved for future use, and may be added to the declarations in the **<complex.h>** header.

cerf( ) *cexpm1( )* *clog2( )*
cerfc( ) *clog10( )* *clgamma( )*
cexp2( ) *clog1p( )* *ctgamma( )*

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *cabs*( ), *cacos*( ), *cacosh*( ), *carg*( ), *casin*( ), *casinh*( ), *catan*( ), *catanh*( ), *ccos*( ), *ccosh*( ), *cexp*( ), *cimag*( ), *clog*( ), *conj*( ), *cpow*( ), *cproj*( ), *creal*( ), *csin*( ), *csinh*( ), *csqrt*( ), *ctan*( ), *ctanh*( )

CHANGE HISTORY

First released in Issue 6. Included for alignment with the ISO / IEC 9899: 1999 standard.

NAME

cpio.h — cpio archive values

SYNOPSIS

XSI

```
#include <cpio.h>
```

DESCRIPTION

Values needed by the *c_mode* field of the *cpio* archive format are described as follows:

Name	Description	Value (Octal)
C_IRUSR	Read by owner.	0000400
C_IWUSR	Write by owner.	0000200
C_IXUSR	Execute by owner.	0000100
C_IRGRP	Read by group.	0000040
C_IWGRP	Write by group.	0000020
C_IXGRP	Execute by group.	0000010
C_IROTH	Read by others.	0000004
C_IWOTH	Write by others.	0000002
C_IXOTH	Execute by others.	0000001
C_ISUID	Set user ID.	0004000
C_ISGID	Set group ID.	0002000
C_ISVTX	On directories, restricted deletion flag.	0001000
C_ISDIR	Directory.	0040000
C_ISFIFO	FIFO.	0010000
C_ISREG	Regular file.	0100000
C_ISBLK	Block special.	0060000
C_ISCHR	Character special.	0020000
C_ISCTG	Reserved.	0110000
C_ISLNK	Symbolic link.	0120000
C_ISSOCK	Socket.	0140000

The header shall define the symbolic constant:

```
MAGIC    "070707"
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Shell and Utilities volume of IEEE Std 1003.1-200x, *pax*

CHANGE HISTORY

First released in Issue 3 of the Headers Interface, Issue 3 specification. Derived from the POSIX.1-1988 standard.

Issue 6

The SEE ALSO is updated to refer to *pax*, since the *cpio* utility is not included in the Shell and Utilities volume of IEEE Std 1003.1-200x.

NAME

ctype.h — character types

SYNOPSIS

```
#include <ctype.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
      int   isalnum(int);
      int   isalpha(int);
XSI   int   isascii(int);
      int   isblank(int);
      int   iscntrl(int);
      int   isdigit(int);
      int   isgraph(int);
      int   islower(int);
      int   isprint(int);
      int   ispunct(int);
      int   isspace(int);
      int   isupper(int);
      int   isxdigit(int);
XSI   int   toascii(int);
      int   tolower(int);
      int   toupper(int);
```

The following are defined as macros:

```
XSI   int   _toupper(int);
      int   _tolower(int);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<locale.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *isalnum*( ), *isalpha*( ), *isascii*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *setlocale*( ), *toascii*( ), *tolower*( ), *_tolower*( ), *toupper*( ), *_toupper*( ), *wcstombs*( ), *wctomb*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

dirent.h — format of directory entries

SYNOPSIS

```
#include <dirent.h>
```

DESCRIPTION

The internal format of directories is unspecified.

The **<dirent.h>** header shall define the following type:

DIR A type representing a directory stream.

It shall also define the structure **dirent** which shall include the following members:

XSI
```
ino_t   d_ino         File serial number.
char    d_name[]      Name of entry.
```

XSI The type **ino_t** shall be defined as described in **<sys/types.h>**.

The character array *d_name* is of unspecified size, but the number of bytes preceding the terminating null byte shall not exceed {NAME_MAX}.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
int            closedir(DIR *);
DIR           *opendir(const char *);
struct dirent *readdir(DIR *);
int            readdir_r(DIR *restrict, struct dirent *restrict,
                   struct dirent **restrict);
void           rewinddir(DIR *);
void           seekdir(DIR *, long);
long           telldir(DIR *);
```

TSF (readdir_r); XSI (seekdir, telldir)

APPLICATION USAGE

None.

RATIONALE

Information similar to that in the **<dirent.h>** header is contained in a file **<sys/dir.h>** in 4.2 BSD and 4.3 BSD. The equivalent in these implementations of **struct dirent** from this volume of IEEE Std 1003.1-200x is **struct direct**. The filename was changed because the name **<sys/dir.h>** was also used in earlier implementations to refer to definitions related to the older access method; this produced name conflicts. The name of the structure was changed because this volume of IEEE Std 1003.1-200x does not completely define what is in the structure, so it could be different on some implementations from **struct direct**.

The name of an array of **char** of an unspecified size should not be used as an lvalue. Use of: |

```
sizeof(d_name)
```

is incorrect; use:

```
strlen(d_name)
```

instead.

The array of **char** *d_name* is not a fixed size. Implementations may need to declare **struct dirent** with an array size for *d_name* of 1, but the actual number of characters provided matches (or only slightly exceeds) the length of the filename.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *closedir*( ), *opendir*( ), *readdir*( ), *readdir_r*( ), *rewinddir*( ), *seekdir*( ), *telldir*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r*( ).

The **restrict** keyword is added to the prototype for *readdir_r*( ).

NAME

dlfcn.h — dynamic linking

SYNOPSIS

XSI
```
#include <dlfcn.h>
```

DESCRIPTION

The **<dlfcn.h>** header shall define at least the following macros for use in the construction of a *dlopen*( ) *mode* argument:

RTLD_LAZY	Relocations are performed at an implementation-defined time.
RTLD_NOW	Relocations are performed when the object is loaded.
RTLD_GLOBAL	All symbols are available for relocation processing of other modules.
RTLD_LOCAL	All symbols are not made available for relocation processing by other modules.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
int    dlclose(void *);
char  *dlerror(void);
void  *dlopen(const char *, int);
void  *dlsym(void *restrict, const char *restrict);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *dlopen*( ), *dlclose*( ), *dlsym*( ), *dlerror*( )

CHANGE HISTORY

First released in Issue 5.

Issue 6

The **restrict** keyword is added to the prototype for *dlsym*( ).

NAME

errno.h — system error numbers

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The ISO C standard only requires the symbols [EDOM], [EILSEQ], and [ERANGE] to be defined.

The **<errno.h>** header shall provide a declaration for *errno* and give positive values for the following symbolic constants. Their values shall be unique except as noted below:

[E2BIG]	Argument list too long.
[EACCES]	Permission denied.
[EADDRINUSE]	Address in use.
[EADDRNOTAVAIL]	Address not available.
[EAFNOSUPPORT]	Address family not supported.
[EAGAIN]	Resource unavailable, try again (may be the same value as [EWOULDBLOCK]).
[EALREADY]	Connection already in progress.
[EBADF]	Bad file descriptor.
[EBADMSG]	Bad message.
[EBUSY]	Device or resource busy.
[ECANCELED]	Operation canceled.
[ECHILD]	No child processes.
[ECONNABORTED]	Connection aborted.
[ECONNREFUSED]	Connection refused.
[ECONNRESET]	Connection reset.
[EDEADLK]	Resource deadlock would occur.
[EDESTADDRREQ]	Destination address required.
[EDOM]	Mathematics argument out of domain of function.
[EDQUOT]	Reserved.
[EEXIST]	File exists.
[EFAULT]	Bad address.
[EFBIG]	File too large.
[EHOSTUNREACH]	Host is unreachable.
[EIDRM]	Identifier removed.
[EILSEQ]	Illegal byte sequence.

	[EINPROGRESS]	Operation in progress.
	[EINTR]	Interrupted function.
	[EINVAL]	Invalid argument.
	[EIO]	I/O error.
	[EISCONN]	Socket is connected.
	[EISDIR]	Is a directory.
	[ELOOP]	Too many levels of symbolic links.
	[EMFILE]	Too many open files.
	[EMLINK]	Too many links.
	[EMSGSIZE]	Message too large.
	[EMULTIHOP]	Reserved.
	[ENAMETOOLONG]	Filename too long.
	[ENETDOWN]	Network is down.
	[ENETUNREACH]	Network unreachable.
	[ENFILE]	Too many files open in system.
	[ENOBUFS]	No buffer space available.
XSR	[ENODATA]	No message is available on the STREAM head read queue.
	[ENODEV]	No such device.
	[ENOENT]	No such file or directory.
	[ENOEXEC]	Executable file format error.
	[ENOLCK]	No locks available.
	[ENOLINK]	Reserved.
	[ENOMEM]	Not enough space.
	[ENOMSG]	No message of the desired type.
	[ENOPROTOOPT]	Protocol not available.
	[ENOSPC]	No space left on device.
XSR	[ENOSR]	No STREAM resources.
XSR	[ENOSTR]	Not a STREAM.
	[ENOSYS]	Function not supported.
	[ENOTCONN]	The socket is not connected.
	[ENOTDIR]	Not a directory.
	[ENOTEMPTY]	Directory not empty.
	[ENOTSOCK]	Not a socket.
	[ENOTSUP]	Not supported.

[ENOTTY]	Inappropriate I/O control operation.
[ENXIO]	No such device or address.
[EOPNOTSUPP]	Operation not supported on socket.
[EOVERFLOW]	Value too large to be stored in data type.
[EPERM]	Operation not permitted.
[EPIPE]	Broken pipe.
[EPROTO]	Protocol error.
[EPROTONOSUPPORT]	Protocol not supported.
[EPROTOTYPE]	Protocol wrong type for socket.
[ERANGE]	Result too large.
[EROFS]	Read-only file system.
[ESPIPE]	Invalid seek.
[ESRCH]	No such process.
[ESTALE]	Reserved.
XSR [ETIME]	Stream *ioctl*( ) timeout.
[ETIMEDOUT]	Connection timed out.
[ETXTBSY]	Text file busy.
[EWOULDBLOCK]	Operation would block (may be the same value as [EAGAIN]).
[EXDEV]	Cross-device link.

APPLICATION USAGE

Additional error numbers may be defined on conforming systems; see the System Interfaces volume of IEEE Std 1003.1-200x.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, Section 2.3, Error Numbers

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Updated for alignment with the POSIX Realtime Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The majority of the error conditions previously marked as extensions are now mandatory, except for the STREAMS-related error conditions.

Values for *errno* are now required to be distinct positive values rather than non-zero values. This change is for alignment with the ISO/IEC 9899:1999 standard.

NAME

fcntl.h — file control options

SYNOPSIS

```
#include <fcntl.h>
```

DESCRIPTION

The **<fcntl.h>** header shall define the following requests and arguments for use by the functions *fcntl*( ) and *open*( ).

Values for *cmd* used by *fcntl*( ) (the following values are unique) are as follows:

F_DUPFD	Duplicate file descriptor.
F_GETFD	Get file descriptor flags.
F_SETFD	Set file descriptor flags.
F_GETFL	Get file status flags and file access modes.
F_SETFL	Set file status flags.
F_GETLK	Get record locking information.
F_SETLK	Set record locking information.
F_SETLKW	Set record locking information; wait if blocked.
F_GETOWN	Get process or process group ID to receive SIGURG signals.
F_SETOWN	Set process or process group ID to receive SIGURG signals.

File descriptor flags used for *fcntl*( ) are as follows:

FD_CLOEXEC	Close the file descriptor upon execution of an *exec* family function.

Values for *l_type* used for record locking with *fcntl*( ) (the following values are unique) are as follows:

F_RDLCK	Shared or read lock.
F_UNLCK	Unlock.
F_WRLCK	Exclusive or write lock.

XSI The values used for *l_whence*, SEEK_SET, SEEK_CUR, and SEEK_END shall be defined as described in **<unistd.h>**.

The following values are file creation flags and are used in the *oflag* value to *open*( ). They shall be bitwise-distinct.

O_CREAT	Create file if it does not exist.
O_EXCL	Exclusive use flag.
O_NOCTTY	Do not assign controlling terminal.
O_TRUNC	Truncate flag.

File status flags used for *open*( ) and *fcntl*( ) are as follows:

	O_APPEND	Set append mode.
SIO	O_DSYNC	Write according to synchronized I/O data integrity completion.
	O_NONBLOCK	Non-blocking mode.

SIO O_RSYNC	Synchronized read I/O operations.
O_SYNC	Write according to synchronized I/O file integrity completion.

Mask for use with file access modes is as follows:

O_ACCMODE	Mask for file access modes.

File access modes used for *open*( ) and *fcntl*( ) are as follows:

O_RDONLY	Open for reading only.
O_RDWR	Open for reading and writing.
O_WRONLY	Open for writing only.

XSI The symbolic names for file modes for use as values of **mode_t** shall be defined as described in **<sys/stat.h>**.

ADV Values for *advice* used by *posix_fadvise*( ) are as follows:

POSIX_FADV_NORMAL
The application has no advice to give on its behavior with respect to the specified data. It is the default characteristic if no advice is given for an open file.

POSIX_FADV_SEQUENTIAL
The application expects to access the specified data sequentially from lower offsets to higher offsets.

POSIX_FADV_RANDOM
The application expects to access the specified data in a random order.

POSIX_FADV_WILLNEED
The application expects to access the specified data in the near future.

POSIX_FADV_DONTNEED
The application expects that it will not access the specified data in the near future.

POSIX_FADV_NOREUSE
The application expects to access the specified data once and then not reuse it thereafter.

The structure **flock** describes a file lock. It shall include the following members:

`short`	`l_type`	Type of lock; F_RDLCK, F_WRLCK, F_UNLCK.
`short`	`l_whence`	Flag for starting offset.
`off_t`	`l_start`	Relative offset in bytes.
`off_t`	`l_len`	Size; if 0 then until EOF.
`pid_t`	`l_pid`	Process ID of the process holding the lock; returned with F_GETLK.

The **mode_t**, **off_t**, and **pid_t** types shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int  creat(const char *, mode_t);
int  fcntl(int, int, ...);
int  open(const char *, int, ...);
int  posix_fadvise(int, off_t, size_t, int);
int  posix_fallocate(int, off_t, size_t);
```

ADV

XSI Inclusion of the **<fcntl.h>** header may also make visible all symbols from **<sys/stat.h>** and **<unistd.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/stat.h>, **<sys/types.h>**, **<unistd.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *creat*( ), *exec*( ), *fcntl*( ), *open*( ), *posix_fadvise*( ), *posix_fallocate*( ), *posix_madvise*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with POSIX Realtime Extension.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- O_DSYNC and O_RSYNC are marked as part of the Synchronized Input and Output option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The definition of the **mode_t**, **off_t**, and **pid_t** types is mandated.

The F_GETOWN and F_SETOWN values are added for sockets.

The *posix_fadvise*( ), *posix_fallocate*( ), and *posix_madvise*( ) functions are added for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #102 is applied moving the prototype for *posix_madvise*( ) to **<sys/mman.h>**. |

NAME

fenv.h — floating-point environment

SYNOPSIS

```
#include <fenv.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<fenv.h>** header shall define the following data types through **typedef**:

fenv_t	Represents the entire floating-point environment. The floating-point environment refers collectively to any floating-point status flags and control modes supported by the implementation.
fexcept_t	Represents the floating-point status flags collectively, including any status the implementation associates with the flags. A floating-point status flag is a system variable whose value is set (but never cleared) when a floating-point exception is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information. A floating-point control mode is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

The **<fenv.h>** header shall define the following constants if and only if the implementation supports the floating-point exception by means of the floating-point functions *fwclearexcept*( ), *fegetexceptflag*( ), *feraiseexcept*( ), *fesetexceptflag*( ), and *fetestexcept*( ). Each expands to an integer constant expression with values such that bitwise-inclusive ORs of all combinations of the constants result in distinct values.

FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW

The **<fenv.h>** header shall define the following constant, which is simply the bitwise-inclusive OR of all floating-point exception constants defined above:

FE_ALL_EXCEPT

The **<fenv.h>** header shall define the following constants if and only if the implementation supports getting and setting the represented rounding direction by means of the *fegetround*( ) and *fesetround*( ) functions. Each expands to an integer constant expression whose values are distinct non-negative vales.

FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD

The **<fenv.h>** header shall define the following constant, which represents the default floating-point environment (that is, the one installed at program startup) and has type pointer to const-qualified **fenv_t**. It can be used as an argument to the functions within the **<fenv.h>** header that manage the floating-point environment.

FE_DFL_ENV

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int  feclearexcept(int);
int  fegetexceptflag(fexcept_t *, int);
int  feraiseexcept(int);
int  fesetexceptflag(const fexcept_t *, int);
int  fetestexcept(int);
int  fegetround(void);
int  fesetround(int);
int  fegetenv(fenv_t *);
int  feholdexcept(fenv_t *);
int  fesetenv(const fenv_t *);
int  feupdateenv(const fenv_t *);
```

The FENV_ACCESS pragma provides a means to inform the implementation when an application might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes. The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another FENV_ACCESS pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another FENV_ACCESS pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of an application tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the FENV_ACCESS pragma off, the behavior is undefined. The default state (on or off) for the pragma is implementation-defined. (When execution passes from a part of the application translated with FENV_ACCESS off to a part translated with FENV_ACCESS on, the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

APPLICATION USAGE

This header is designed to support the floating-point exception status flags and directed-rounding control modes required by the IEC 60559: 1989 standard, and other similar floating-point state information. Also it is designed to facilitate code portability among all systems.

Certain application programming conventions support the intended model of use for the floating-point environment:

- A function call does not alter its caller's floating-point control modes, clear its caller's floating-point status flags, nor depend on the state of its caller's floating-point status flags unless the function is so documented.
- A function call is assumed to require default floating-point control modes, unless its documentation promises otherwise.
- A function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

With these conventions, an application can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the application that does so explicitly.

Even though the rounding direction macros may expand to constants corresponding to the values of FLT_ROUNDS, they are not required to do so.

For example:

```
#include <fenv.h>
void f(double x)
{
    #pragma STDC FENV_ACCESS ON
    void g(double);
    void h(double);
    /* ... */
    g(x + 1);
    h(x + 1);
    /* ... */
}
```

If the function *g*( ) might depend on status flags set as a side effect of the first *x*+1, or if the second *x*+1 might depend on control modes set as a side effect of the call to function *g*( ), then the application shall contain an appropriately placed invocation as follows:

```
#pragma STDC FENV_ACCESS ON
```

RATIONALE

The fexcept_t Type

fexcept_t does not have to be an integer type. Its values must be obtained by a call to *fegetexceptflag*( ), and cannot be created by logical operations from the exception macros. An implementation might simply implement **fexcept_t** as an **int** and use the representations reflected by the exception macros, but is not required to; other representations might contain extra information about the exceptions. **fexcept_t** might be a **struct** with a member for each exception (that might hold the address of the first or last floating-point instruction that caused that exception). The ISO/IEC 9899:1999 standard makes no claims about the internals of an **fexcept_t**, and so the user cannot inspect it.

Exception and Rounding Macros

Macros corresponding to unsupported modes and rounding directions are not defined by the implementation and must not be defined by the application. An application might use **#ifdef** to test for this. |

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *feclearexcept*( ), *fegetenv*( ), *fegetexceptflag*( ), *fegetround*( ), *feholdexcept*( ), *feraiseexcept*( ), *fesetenv*( ), *fesetexceptflag*( ), *fesetround*( ), *fetestexcept*( ), *feupdateenv*( )

CHANGE HISTORY

First released in Issue 6. Included for alignment with the ISO/IEC 9899:1999 standard.

The return types for *feclearexcept*( ), *fegetexcepflag*( ), *feraiseexcept*( ), *fesetexceptflag*( ), *fegetenv*( ), *fesetenv*( ), and *feupdateenv*( ) are changed from **void** to **int** for alignment with the ISO/IEC 9899:1999 standard, Defect Report 202.

NAME

float.h — floating types

SYNOPSIS

```
#include <float.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.

The following parameters are used to define the model for each floating-point type:

s	Sign (±1).
b	Base or radix of exponent representation (an integer >1).
e	Exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$).
p	Precision (the number of base–b digits in the significand).
f_k	Non-negative integers less than b (the significand digits).

A floating-point number x is defined by the following model:

$$x = sb^{e} \sum_{k=1}^{p} f_k\, b^{-k},\ e_{\min} \le e \le e_{\max}$$

In addition to normalized floating-point numbers (f_1>0 if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers ($x \neq 0$, $e = e_{\min}$, $f_1 = 0$) and unnormalized floating-point numbers ($x \neq 0$, $e > e_{\min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand.

The accuracy of the floating-point operations (`'+'`, `'−'`, `'*'`, `'/'`) and of the library functions in **<math.h>** and **<complex.h>** that return floating-point results is implementation-defined. The implementation may state that the accuracy is unknown.

All integer values in the **<float.h>** header, except FLT_ROUNDS, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. All except DECIMAL_DIG, FLT_EVAL_METHOD, FLT_RADIX, and FLT_ROUNDS have separate names for all three floating-point types. The floating-point model representation is provided for all values except FLT_EVAL_METHOD and FLT_ROUNDS.

The rounding mode for floating-point addition is characterized by the implementation-defined value of FLT_ROUNDS:

−1	Indeterminable.
0	Toward zero.
1	To nearest.
2	Toward positive infinity.

3 Toward negative infinity.

All other values for FLT_ROUNDS characterize implementation-defined rounding behavior.

The values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of FLT_EVAL_METHOD:

−1 Indeterminable.

0 Evaluate all operations and constants just to the range and precision of the type.

1 Evaluate operations and constants of type **float** and **double** to the range and precision of the **double** type, evaluate **long double** operations and constants to the range and precision of the **long double** type.

2 Evaluate all operations and constants to the range and precision of the **long double** type.

All other negative values for FLT_EVAL_METHOD characterize implementation-defined behavior.

The values given in the following list shall be defined as constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign.

- Radix of exponent representation, *b*.

  FLT_RADIX 2

- Number of base-FLT_RADIX digits in the floating-point significand, *p*.

  FLT_MANT_DIG

  DBL_MANT_DIG

  LDBL_MANT_DIG

- Number of decimal digits, *n*, such that any floating-point number in the widest supported floating type with $p_{\max}$ radix *b* digits can be rounded to a floating-point number with *n* decimal digits and back again without change to the value.

$$\begin{cases} p_{\max} \log_{10} b & \textit{if } b \textit{ is a power of } 10 \\ \lceil 1 + p_{\max} \log_{10} b \rceil & \textit{otherwise} \end{cases}$$

  DECIMAL_DIG 10

- Number of decimal digits, *q*, such that any floating-point number with *q* decimal digits can be rounded into a floating-point number with *p* radix *b* digits and back again without change to the *q* decimal digits.

$$\begin{cases} p \log_{10} b & \textit{if } b \textit{ is a power of } 10 \\ \lfloor (p-1) \log_{10} b \rfloor & \textit{otherwise} \end{cases}$$

  FLT_DIG 6

  DBL_DIG 10

LDBL_DIG	10

- Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, $e_{\min}$.

FLT_MIN_EXP	
DBL_MIN_EXP	
LDBL_MIN_EXP	

- Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.

$$\left\lceil \log_{10} b^{e_{\min}-1} \right\rceil$$

FLT_MIN_10_EXP	–37
DBL_MIN_10_EXP	–37
LDBL_MIN_10_EXP	–37

- Maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, $e_{\max}$.

FLT_MAX_EXP	
DBL_MAX_EXP	
LDBL_MAX_EXP	

- Maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.

$$\left\lfloor \log_{10}((1-b^{-p})\, b^{e_{\max}}) \right\rfloor$$

FLT_MAX_10_EXP	+37
DBL_MAX_10_EXP	+37
LDBL_MAX_10_EXP	+37

The values given in the following list shall be defined as constant expressions with implementation-defined values that are greater than or equal to those shown:

- Maximum representable finite floating-point number.

$$(1-b^{-p})\, b^{e_{\max}}$$

FLT_MAX	1E+37
DBL_MAX	1E+37
LDBL_MAX	1E+37

The values given in the following list shall be defined as constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

- The difference between 1 and the least value greater that 1 that is representable in the given floating-point type, b^{1-p}.

FLT_EPSILON	1E–5
DBL_EPSILON	1E–9

LDBL_EPSILON	1E–9

- Minimum normalized positive floating-point number, $b^{e_{\min}-1}$.

FLT_MIN	1E–37
DBL_MIN	1E–37
LDBL_MIN	1E–37

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<complex.h>, **<math.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

Issue 6

The description of the operations with floating-point values is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fmtmsg.h — message display structures

SYNOPSIS

XSI `#include <fmtmsg.h>`

DESCRIPTION

The **<fmtmsg.h>** header shall define the following macros, which expand to constant integer expressions:

MM_HARD	Source of the condition is hardware.
MM_SOFT	Source of the condition is software.
MM_FIRM	Source of the condition is firmware.
MM_APPL	Condition detected by application.
MM_UTIL	Condition detected by utility.
MM_OPSYS	Condition detected by operating system.
MM_RECOVER	Recoverable error.
MM_NRECOV	Non-recoverable error.
MM_HALT	Error causing application to halt.
MM_ERROR	Application has encountered a non-fatal fault.
MM_WARNING	Application has detected unusual non-error condition.
MM_INFO	Informative message.
MM_NOSEV	No severity level provided for the message.
MM_PRINT	Display message on standard error.
MM_CONSOLE	Display message on system console.

The table below indicates the null values and identifiers for *fmtmsg*( ) arguments. The **<fmtmsg.h>** header shall define the macros in the **Identifier** column, which expand to constant expressions that expand to expressions of the type indicated in the **Type** column:

Argument	Type	Null-Value	Identifier
label	**char** *	(**char***)0	MM_NULLLBL
severity	**int**	0	MM_NULLSEV
class	**long**	**0L**	MM_NULLMC
text	**char** *	(**char***)0	MM_NULLTXT
action	**char** *	(**char***)0	MM_NULLACT
tag	**char** *	(**char***)0	MM_NULLTAG

The **<fmtmsg.h>** header shall also define the following macros for use as return values for *fmtmsg*( ):

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.

MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided.

```
int fmtmsg(long, const char *, int,
    const char *, const char *, const char *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *fmtmsg*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

fnmatch.h — filename-matching types

SYNOPSIS

```
#include <fnmatch.h>
```

DESCRIPTION

The **<fnmatch.h>** header shall define the following constants:

FNM_NOMATCH	The string does not match the specified pattern.
FNM_PATHNAME	Slash in *string* only matches slash in *pattern*.
FNM_PERIOD	Leading period in *string* must be exactly matched by period in *pattern*.
FNM_NOESCAPE	Disable backslash escaping.
OB XSI FNM_NOSYS	Reserved.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided. |

```
int fnmatch(const char *, const char *, int);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *fnmatch*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 6

The constant FNM_NOSYS is marked obsolescent.

NAME

ftw.h — file tree traversal

SYNOPSIS

XSI
```
#include <ftw.h>
```

DESCRIPTION

The **<ftw.h>** header shall define the **FTW** structure that includes at least the following members:

```
int  base
int  level
```

The **<ftw.h>** header shall define macros for use as values of the third argument to the application-supplied function that is passed as the second argument to *ftw*( ) and *nftw*( ):

FTW_F	File.
FTW_D	Directory.
FTW_DNR	Directory without read permission.
FTW_DP	Directory with subdirectories visited.
FTW_NS	Unknown type; *stat*( ) failed.
FTW_SL	Symbolic link.
FTW_SLN	Symbolic link that names a nonexistent file.

The **<ftw.h>** header shall define macros for use as values of the fourth argument to *nftw*( ):

FTW_PHYS	Physical walk, does not follow symbolic links. Otherwise, *nftw*( ) follows links but does not walk down any path that crosses itself.
FTW_MOUNT	The walk does not cross a mount point.
FTW_DEPTH	All subdirectories are visited before the directory itself.
FTW_CHDIR	The walk changes to each directory before reading it.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int ftw(const char *,
    int (*)(const char *, const struct stat *, int), int);
int nftw(const char *, int (*)
    (const char *, const struct stat *, int, struct FTW*),
    int, int);
```

The **<ftw.h>** header shall define the **stat** structure and the symbolic names for *st_mode* and the file type test macros as described in **<sys/stat.h>**.

Inclusion of the **<ftw.h>** header may also make visible all symbols from **<sys/stat.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/stat.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *ftw*( ), *nftw*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A description of FTW_DP is added.

<glob.h>

NAME

glob.h — pathname pattern-matching types

SYNOPSIS

```
#include <glob.h>
```

DESCRIPTION

The **<glob.h>** header shall define the structures and symbolic constants used by the *glob*( ) function.

The structure type **glob_t** shall contain at least the following members:

`size_t`	`gl_pathc`	Count of paths matched by *pattern*.
`char`	`**gl_pathv`	Pointer to a list of matched pathnames.
`size_t`	`gl_offs`	Slots to reserve at the beginning of *gl_pathv*.

The following constants shall be provided as values for the *flags* argument:

GLOB_APPEND	Append generated pathnames to those previously obtained.
GLOB_DOOFFS	Specify how many null pointers to add to the beginning of *pglob->gl_pathv*.
GLOB_ERR	Cause *glob*( ) to return on error.
GLOB_MARK	Each pathname that is a directory that matches *pattern* has a slash appended.
GLOB_NOCHECK	If *pattern* does not match any pathname, then return a list consisting of only *pattern*.
GLOB_NOESCAPE	Disable backslash escaping.
GLOB_NOSORT	Do not sort the pathnames returned.

The following constants shall be defined as error return values:

GLOB_ABORTED	The scan was stopped because GLOB_ERR was set or (**errfunc*)( ) returned non-zero.
GLOB_NOMATCH	The pattern does not match any existing pathname, and GLOB_NOCHECK was not set in flags.
GLOB_NOSPACE	An attempt to allocate memory failed.
OB XSI GLOB_NOSYS	Reserved.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int  glob(const char *restrict, int, int (*restrict)(const char *, int),
          glob_t *restrict);
void globfree (glob_t *);
```

The implementation may define additional macros or constants using names beginning with GLOB_.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *glob*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 6

The **restrict** keyword is added to the prototype for *glob*( ).

The constant GLOB_NOSYS is marked obsolescent.

NAME

grp.h — group structure

SYNOPSIS

```
#include <grp.h>
```

DESCRIPTION

The **<grp.h>** header shall declare the structure **group** which shall include the following members:

`char`	`*gr_name`	The name of the group.
`gid_t`	`gr_gid`	Numerical group ID.
`char`	`**gr_mem`	Pointer to a null-terminated array of character pointers to member names.

The **gid_t** type shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
struct group  *getgrgid(gid_t);
struct group  *getgrnam(const char *);
```

TSF

```
int            getgrgid_r(gid_t, struct group *, char *,
                   size_t, struct group **);
int            getgrnam_r(const char *, struct group *, char *,
                   size_t , struct group **);
```

XSI

```
struct group  *getgrent(void);
void           endgrent(void);
void           setgrent(void);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *endgrent*( ), *getgrgid*( ), *getgrnam*( )

CHANGE HISTORY

First released in Issue 1.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The definition of **gid_t** is mandated.
- The *getgrgid_r*( ) and *getgrnam_r*( ) functions are marked as part of the Thread-Safe Functions option.

NAME

iconv.h — codeset conversion facility

SYNOPSIS

XSI
```
#include <iconv.h>
```

DESCRIPTION

The **<iconv.h>** header shall define the following type:

iconv_t	Identifies the conversion from one codeset to another.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
iconv_t iconv_open(const char *, const char *);
size_t  iconv(iconv_t, char **restrict, size_t *restrict, char **restrict,
            size_t *restrict);
int     iconv_close(iconv_t);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *iconv*( ), *iconv_close*( ), *iconv_open*( )

CHANGE HISTORY

First released in Issue 4.

Issue 6

The **restrict** keyword is added to the prototype for *iconv*( ).

NAME

inttypes.h — fixed size integer types

SYNOPSIS

```
#include <inttypes.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<inttypes.h>** header shall include the **<stdint.h>** header.

The **<inttypes.h>** header shall include a definition of at least the following type:

imaxdiv_t Structure type that is the type of the value returned by the *imaxdiv*( ) function.

The following macros shall be defined. Each expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the *format* argument of a formatted input/output function when converting the corresponding integer type. These macros have the general form of PRI (character string literals for the *fprintf*( ) and *fwprintf*( ) family of functions) or SCN (character string literals for the *fscanf*( ) and *fwscanf*( ) family of functions), followed by the conversion specifier, followed by a name corresponding to a similar type name in **<stdint.h>**. In these names, *N* represents the width of the type as described in **<stdint.h>**. For example, *PRIdFAST32* can be used in a format string to print the value of an integer of type **int_fast32_t**.

The *fprintf*( ) macros for signed integers are:

PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR

The *fprintf*( ) macros for unsigned integers are:

PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR

The *fscanf*( ) macros for signed integers are:

SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR

The *fscanf*( ) macros for unsigned integers are:

SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR

For each type that the implementation provides in **<stdint.h>**, the corresponding *fprintf*( ) macros shall be defined and the corresponding *fscanf*( ) macros shall be defined unless the implementation does not have a suitable fscanf length modifier for the type.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
intmax_t  imaxabs(intmax_t);
imaxdiv_t imaxdiv(intmax_t, intmax_t);
intmax_t  strtoimax(const char *restrict, char **restrict, int);
```

```
uintmax_t strtoumax(const char *restrict, char **restrict, int);
intmax_t  wcstoimax(const wchar_t *restrict, wchar_t **restrict, int);
uintmax_t wcstoumax(const wchar_t *restrict, wchar_t **restrict, int);
```

EXAMPLES

```
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
    uintmax_t i = UINTMAX_MAX; // This type always exists.
    wprintf(L"The largest integer value is %020"
        PRIxMAX "\n", i);
    return 0;
}
```

APPLICATION USAGE

None.

RATIONALE

The ISO/IEC 9899:1990 standard specifies that the language should support four signed and unsigned integer data types—**char**, **short**, **int**, and **long**—but places very little requirement on their size other than that **int** and **short** be at least 16 bits and **long** be at least as long as **int** and not smaller than 32 bits. For 16-bit systems, most implementations assign 8, 16, 16, and 32 bits to **char**, **short**, **int**, and **long**, respectively. For 32-bit systems, the common practice is to assign 8, 16, 32, and 32 bits to these types. This difference in **int** size can create some problems for users who migrate from one system to another which assigns different sizes to integer types, because the ISO C standard integer promotion rule can produce silent changes unexpectedly. The need for defining an extended integer type increased with the introduction of 64-bit systems.

The purpose of **<inttypes.h>** is to provide a set of integer types whose definitions are consistent across machines and independent of operating systems and other implementation idiosyncrasies. It defines, via **typedef**, integer types of various sizes. Implementations are free to **typedef** them as ISO C standard integer types or extensions that they support. Consistent use of this header will greatly increase the portability of a users program across platforms.

FUTURE DIRECTIONS

Macro names beginning with PRI or SCN followed by any lowercase letter or `'X'` may be added to the macros defined in the **<inttypes.h>** header.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *imaxdiv*( )

CHANGE HISTORY

First released in Issue 5.

Issue 6

The Open Group Base Resolution bwg97-006 is applied.

This reference page is updated to align with the ISO/IEC 9899:1999 standard.

NAME

iso646.h — alternative spellings

SYNOPSIS

```
#include <iso646.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<iso646.h>** header shall define the following eleven macros (on the left) that expand to the corresponding tokens (on the right):

and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 5. Derived from ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

NAME

langinfo.h — language information constants

SYNOPSIS

XSI
```
#include <langinfo.h>
```

DESCRIPTION

The **<langinfo.h>** header contains the constants used to identify items of *langinfo* data (see *nl_langinfo*( )). The type of the constant, **nl_item**, shall be defined as described in **<nl_types.h>**.

The following constants shall be defined. The entries under **Category** indicate in which *setlocale*( ) category each item is defined.

Constant	Category	Meaning
CODESET	*LC_CTYPE*	Codeset name.
D_T_FMT	*LC_TIME*	String for formatting date and time.
D_FMT	*LC_TIME*	Date format string.
T_FMT	*LC_TIME*	Time format string.
T_FMT_AMPM	*LC_TIME*	a.m. or p.m. time format string.
AM_STR	*LC_TIME*	Ante Meridian affix.
PM_STR	*LC_TIME*	Post Meridian affix.
DAY_1	*LC_TIME*	Name of the first day of the week (for example, Sunday).
DAY_2	*LC_TIME*	Name of the second day of the week (for example, Monday).
DAY_3	*LC_TIME*	Name of the third day of the week (for example, Tuesday).
DAY_4	*LC_TIME*	Name of the fourth day of the week (for example, Wednesday).
DAY_5	*LC_TIME*	Name of the fifth day of the week (for example, Thursday).
DAY_6	*LC_TIME*	Name of the sixth day of the week (for example, Friday).
DAY_7	*LC_TIME*	Name of the seventh day of the week (for example, Saturday).
ABDAY_1	*LC_TIME*	Abbreviated name of the first day of the week.
ABDAY_2	*LC_TIME*	Abbreviated name of the second day of the week.
ABDAY_3	*LC_TIME*	Abbreviated name of the third day of the week.
ABDAY_4	*LC_TIME*	Abbreviated name of the fourth day of the week.
ABDAY_5	*LC_TIME*	Abbreviated name of the fifth day of the week.
ABDAY_6	*LC_TIME*	Abbreviated name of the sixth day of the week.
ABDAY_7	*LC_TIME*	Abbreviated name of the seventh day of the week.
MON_1	*LC_TIME*	Name of the first month of the year.
MON_2	*LC_TIME*	Name of the second month.
MON_3	*LC_TIME*	Name of the third month.
MON_4	*LC_TIME*	Name of the fourth month.
MON_5	*LC_TIME*	Name of the fifth month.
MON_6	*LC_TIME*	Name of the sixth month.
MON_7	*LC_TIME*	Name of the seventh month.
MON_8	*LC_TIME*	Name of the eighth month.
MON_9	*LC_TIME*	Name of the ninth month.
MON_10	*LC_TIME*	Name of the tenth month.
MON_11	*LC_TIME*	Name of the eleventh month.
MON_12	*LC_TIME*	Name of the twelfth month.

Constant	Category	Meaning
ABMON_1	*LC_TIME*	Abbreviated name of the first month.
ABMON_2	*LC_TIME*	Abbreviated name of the second month.
ABMON_3	*LC_TIME*	Abbreviated name of the third month.
ABMON_4	*LC_TIME*	Abbreviated name of the fourth month.
ABMON_5	*LC_TIME*	Abbreviated name of the fifth month.
ABMON_6	*LC_TIME*	Abbreviated name of the sixth month.
ABMON_7	*LC_TIME*	Abbreviated name of the seventh month.
ABMON_8	*LC_TIME*	Abbreviated name of the eighth month.
ABMON_9	*LC_TIME*	Abbreviated name of the ninth month.
ABMON_10	*LC_TIME*	Abbreviated name of the tenth month.
ABMON_11	*LC_TIME*	Abbreviated name of the eleventh month.
ABMON_12	*LC_TIME*	Abbreviated name of the twelfth month.
ERA	*LC_TIME*	Era description segments.
ERA_D_FMT	*LC_TIME*	Era date format string.
ERA_D_T_FMT	*LC_TIME*	Era date and time format string.
ERA_T_FMT	*LC_TIME*	Era time format string.
ALT_DIGITS	*LC_TIME*	Alternative symbols for digits.
RADIXCHAR	*LC_NUMERIC*	Radix character.
THOUSEP	*LC_NUMERIC*	Separator for thousands.
YESEXPR	*LC_MESSAGES*	Affirmative response expression.
NOEXPR	*LC_MESSAGES*	Negative response expression.
CRNCYSTR	*LC_MONETARY*	Currency symbol, preceded by '−' if the symbol should appear before the value, '+' if the symbol should appear after the value, or '.' if the symbol should replace the radix character.

If the locale's values for **p_cs_precedes** and **n_cs_precedes** do not match, the value of *nl_langinfo*(CRNCYSTR) is unspecified. |

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided. |

```
char *nl_langinfo(nl_item);
```

Inclusion of the **<langinfo.h>** header may also make visible all symbols from **<nl_types.h>**.

APPLICATION USAGE

Wherever possible, users are advised to use functions compatible with those in the ISO C standard to access items of *langinfo* data. In particular, the *strftime*( ) function should be used to access date and time information defined in category *LC_TIME*. The *localeconv*( ) function should be used to access information corresponding to RADIXCHAR, THOUSEP, and CRNCYSTR.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *nl_langinfo*( ), *localeconv*( ), *strfmon*( ), *strftime*( ), Chapter 7 (on page 119)

CHANGE HISTORY

First released in Issue 2.

Issue 5

The constants YESSTR and NOSTR are marked LEGACY.

Issue 6

The constants YESSTR and NOSTR are removed.

NAME

libgen.h — definitions for pattern matching functions

SYNOPSIS

XSI

```
#include <libgen.h>
```

DESCRIPTION

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
char  *basename(char *);
char  *dirname(char *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *basename*( ), *dirname*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

The function prototypes for *basename*( ) and *dirname*( ) are changed to indicate that the first argument is of type **char *** rather than **const char ***.

Issue 6

The **__loc1** symbol and the *regcmp*( ) and *regex*( ) functions are removed.

NAME

limits.h — implementation-defined constants

SYNOPSIS

```
#include <limits.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

CX Many of the symbols listed here are not defined by the ISO/IEC 9899:1999 standard. Such symbols are not shown as CX shaded.

The **<limits.h>** header shall define various symbolic names. Different categories of names are described below.

The names represent various limits on resources that the implementation imposes on applications.

Implementations may choose any appropriate value for each limit, provided it is not more restrictive than the Minimum Acceptable Values listed below. Symbolic constant names beginning with _POSIX may be found in **<unistd.h>**.

Applications should not assume any particular value for a limit. To achieve maximum portability, an application should not require more resource than the Minimum Acceptable Value quantity. However, an application wishing to avail itself of the full amount of a resource available on an implementation may make use of the value given in **<limits.h>** on that particular implementation, by using the symbolic names listed below. It should be noted, however, that many of the listed limits are not invariant, and at runtime, the value of the limit may differ from those given in this header, for the following reasons:

- The limit is pathname-dependent. |
- The limit differs between the compile and runtime machines.

For these reasons, an application may use the *fpathconf*( ), *pathconf*( ), and *sysconf*( ) functions to determine the actual value of a limit at runtime.

The items in the list ending in _MIN give the most negative values that the mathematical types are guaranteed to be capable of representing. Numbers of a more negative value may be supported on some implementations, as indicated by the **<limits.h>** header on the implementation, but applications requiring such numbers are not guaranteed to be portable to all implementations. For positive constants ending in _MIN, this indicates the minimum acceptable value.

The Minimum Acceptable Value symbol '*' indicates that there is no guaranteed value across all conforming implementations.

Runtime Invariant Values (Possibly Indeterminate)

A definition of one of the symbolic names in the following list shall be omitted from **<limits.h>** on specific implementations where the corresponding value is equal to or greater than the stated minimum, but is unspecified.

This indetermination might depend on the amount of available memory space on a specific instance of a specific implementation. The actual value supported by a specific instance shall be provided by the *sysconf*( ) function.

AIO {AIO_LISTIO_MAX}
Maximum number of I/O operations in a single list I/O call supported by the implementation.
Minimum Acceptable Value: {_POSIX_AIO_LISTIO_MAX}

AIO {AIO_MAX}
Maximum number of outstanding asynchronous I/O operations supported by the implementation.
Minimum Acceptable Value: {_POSIX_AIO_MAX}

AIO {AIO_PRIO_DELTA_MAX}
The maximum amount by which a process can decrease its asynchronous I/O priority level from its own scheduling priority.
Minimum Acceptable Value: 0

{ARG_MAX}
Maximum length of argument to the *exec* functions including environment data.
Minimum Acceptable Value: {_POSIX_ARG_MAX}

XSI {ATEXIT_MAX}
Maximum number of functions that may be registered with *atexit*( ).
Minimum Acceptable Value: 32

{CHILD_MAX}
Maximum number of simultaneous processes per real user ID.
Minimum Acceptable Value: {_POSIX_CHILD_MAX}

TMR {DELAYTIMER_MAX}
Maximum number of timer expiration overruns.
Minimum Acceptable Value: {_POSIX_DELAYTIMER_MAX}

{HOST_NAME_MAX}
Maximum length of a host name (not including the terminating null) as returned from the *gethostname*( ) function.
Minimum Acceptable Value: {_POSIX_HOST_NAME_MAX}

XSI {IOV_MAX}
Maximum number of **iovec** structures that one process has available for use with *readv*( ) or *writev*( ).
Minimum Acceptable Value: {_XOPEN_IOV_MAX}

{LOGIN_NAME_MAX}
Maximum length of a login name.
Minimum Acceptable Value: {_POSIX_LOGIN_NAME_MAX}

MSG {MQ_OPEN_MAX}
The maximum number of open message queue descriptors a process may hold.
Minimum Acceptable Value: {_POSIX_MQ_OPEN_MAX}

MSG {MQ_PRIO_MAX}
The maximum number of message priorities supported by the implementation.
Minimum Acceptable Value: {_POSIX_MQ_PRIO_MAX}

{OPEN_MAX}
Maximum number of files that one process can have open at any one time.
Minimum Acceptable Value: {_POSIX_OPEN_MAX}

{PAGESIZE}
Size in bytes of a page.
Minimum Acceptable Value: 1

XSI {PAGE_SIZE}
Equivalent to {PAGESIZE}. If either {PAGESIZE} or {PAGE_SIZE} is defined, the other is defined with the same value.

THR {PTHREAD_DESTRUCTOR_ITERATIONS}
Maximum number of attempts made to destroy a thread's thread-specific data values on thread exit.
Minimum Acceptable Value: {_POSIX_THREAD_DESTRUCTOR_ITERATIONS}

THR {PTHREAD_KEYS_MAX}
Maximum number of data keys that can be created by a process.
Minimum Acceptable Value: {_POSIX_THREAD_KEYS_MAX}

THR {PTHREAD_STACK_MIN}
Minimum size in bytes of thread stack storage.
Minimum Acceptable Value: 0

THR {PTHREAD_THREADS_MAX}
Maximum number of threads that can be created per process.
Minimum Acceptable Value: {_POSIX_THREAD_THREADS_MAX}

{RE_DUP_MAX}
The number of repeated occurrences of a BRE permitted by the *regexec*( ) and *regcomp*( ) functions when using the interval notation {\(*m*,*n*\}; see Section 9.3.6 (on page 170).
Minimum Acceptable Value: {_POSIX2_RE_DUP_MAX}

RTS {RTSIG_MAX}
Maximum number of realtime signals reserved for application use in this implementation.
Minimum Acceptable Value: {_POSIX_RTSIG_MAX}

SEM {SEM_NSEMS_MAX}
Maximum number of semaphores that a process may have.
Minimum Acceptable Value: {_POSIX_SEM_NSEMS_MAX}

SEM {SEM_VALUE_MAX}
The maximum value a semaphore may have.
Minimum Acceptable Value: {_POSIX_SEM_VALUE_MAX}

RTS {SIGQUEUE_MAX}
Maximum number of queued signals that a process may send and have pending at the receiver(s) at any time.
Minimum Acceptable Value: {_POSIX_SIGQUEUE_MAX}

SS|TSP {SS_REPL_MAX}
The maximum number of replenishment operations that may be simultaneously pending for a particular sporadic server scheduler.
Minimum Acceptable Value: {_POSIX_SS_REPL_MAX}

{STREAM_MAX}

The number of streams that one process can have open at one time. If defined, it has the same value as {FOPEN_MAX} (see **<stdio.h>**).
Minimum Acceptable Value: {_POSIX_STREAM_MAX}

{SYMLOOP_MAX}

Maximum number of symbolic links that can be reliably traversed in the resolution of a pathname in the absence of a loop.
Minimum Acceptable Value: {_POSIX_SYMLOOP_MAX}

TMR {TIMER_MAX}

Maximum number of timers per-process supported by the implementation.
Minimum Acceptable Value: {_POSIX_TIMER_MAX}

TRC {TRACE_EVENT_NAME_MAX}

Maximum length of the trace event name.
Minimum Acceptable Value: {_POSIX_TRACE_EVENT_NAME_MAX}

TRC {TRACE_NAME_MAX}

Maximum length of the trace generation version string or of the trace stream name.
Minimum Acceptable Value: {_POSIX_TRACE_NAME_MAX}

TRC {TRACE_SYS_MAX}

Maximum number of trace streams that may simultaneously exist in the system.
Minimum Acceptable Value: {_POSIX_TRACE_SYS_MAX}

TRC {TRACE_USER_EVENT_MAX}

Maximum number of user trace event type identifiers that may simultaneously exist in a traced process, including the predefined user trace event POSIX_TRACE_UNNAMED_USER_EVENT.
Minimum Acceptable Value: {_POSIX_TRACE_USER_EVENT_MAX}

{TTY_NAME_MAX}

Maximum length of terminal device name.
Minimum Acceptable Value: {_POSIX_TTY_NAME_MAX}

{TZNAME_MAX}

Maximum number of bytes supported for the name of a timezone (not of the TZ variable).
Minimum Acceptable Value: {_POSIX_TZNAME_MAX}

Note: The length given by {TZNAME_MAX} does not include the quoting characters mentioned in Section 8.3 (on page 161).

Pathname Variable Values

The values in the following list may be constants within an implementation or may vary from one pathname to another. For example, file systems or directories may have different characteristics.

A definition of one of the values shall be omitted from the **<limits.h>** header on specific implementations where the corresponding value is equal to or greater than the stated minimum, but where the value can vary depending on the file to which it is applied. The actual value supported for a specific pathname shall be provided by the *pathconf*( ) function.

{FILESIZEBITS}

Minimum number of bits needed to represent, as a signed integer value, the maximum size of a regular file allowed in the specified directory.
Minimum Acceptable Value: 32

{LINK_MAX}
Maximum number of links to a single file.
Minimum Acceptable Value: {_POSIX_LINK_MAX}

{MAX_CANON}
Maximum number of bytes in a terminal canonical input line.
Minimum Acceptable Value: {_POSIX_MAX_CANON}

{MAX_INPUT}
Minimum number of bytes for which space is available in a terminal input queue; therefore, the maximum number of bytes a conforming application may require to be typed as input before reading them.
Minimum Acceptable Value: {_POSIX_MAX_INPUT}

{NAME_MAX}
Maximum number of bytes in a filename (not including terminating null).
Minimum Acceptable Value: {_POSIX_NAME_MAX}
XSI Minimum Acceptable Value: {_XOPEN_NAME_MAX}

{PATH_MAX}
Maximum number of bytes in a pathname, including the terminating null character.
Minimum Acceptable Value: {_POSIX_PATH_MAX}
XSI Minimum Acceptable Value: {_XOPEN_PATH_MAX}

{PIPE_BUF}
Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.
Minimum Acceptable Value: {_POSIX_PIPE_BUF}

ADV {POSIX_ALLOC_SIZE_MIN}
Minimum number of bytes of storage actually allocated for any portion of a file.
Minimum Acceptable Value: Not specified.

ADV {POSIX_REC_INCR_XFER_SIZE}
Recommended increment for file transfer sizes between the {POSIX_REC_MIN_XFER_SIZE} and {POSIX_REC_MAX_XFER_SIZE} values.
Minimum Acceptable Value: Not specified.

ADV {POSIX_REC_MAX_XFER_SIZE}
Maximum recommended file transfer size.
Minimum Acceptable Value: Not specified.

ADV {POSIX_REC_MIN_XFER_SIZE}
Minimum recommended file transfer size.
Minimum Acceptable Value: Not specified.

ADV {POSIX_REC_XFER_ALIGN}
Recommended file transfer buffer alignment.
Minimum Acceptable Value: Not specified.

{SYMLINK_MAX}
Maximum number of bytes in a symbolic link.
Minimum Acceptable Value: {_POSIX_SYMLINK_MAX}

Runtime Increasable Values

The magnitude limitations in the following list shall be fixed by specific implementations. An application should assume that the value supplied by **<limits.h>** in a specific implementation is the minimum that pertains whenever the application is run under that implementation. A specific instance of a specific implementation may increase the value relative to that supplied by **<limits.h>** for that implementation. The actual value supported by a specific instance shall be provided by the *sysconf*( ) function.

{BC_BASE_MAX}
Maximum *obase* values allowed by the *bc* utility.
Minimum Acceptable Value: {_POSIX2_BC_BASE_MAX}

{BC_DIM_MAX}
Maximum number of elements permitted in an array by the *bc* utility.
Minimum Acceptable Value: {_POSIX2_BC_DIM_MAX}

{BC_SCALE_MAX}
Maximum *scale* value allowed by the *bc* utility.
Minimum Acceptable Value: {_POSIX2_BC_SCALE_MAX}

{BC_STRING_MAX}
Maximum length of a string constant accepted by the *bc* utility.
Minimum Acceptable Value: {_POSIX2_BC_STRING_MAX}

{CHARCLASS_NAME_MAX}
Maximum number of bytes in a character class name.
Minimum Acceptable Value: {_POSIX2_CHARCLASS_NAME_MAX}

{COLL_WEIGHTS_MAX}
Maximum number of weights that can be assigned to an entry of the *LC_COLLATE* **order** keyword in the locale definition file; see Chapter 7 (on page 119).
Minimum Acceptable Value: {_POSIX2_COLL_WEIGHTS_MAX}

{EXPR_NEST_MAX}
Maximum number of expressions that can be nested within parentheses by the *expr* utility.
Minimum Acceptable Value: {_POSIX2_EXPR_NEST_MAX}

{LINE_MAX}
Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline.
Minimum Acceptable Value: {_POSIX2_LINE_MAX}

{NGROUPS_MAX}
Maximum number of simultaneous supplementary group IDs per process.
Minimum Acceptable Value: {_POSIX_NGROUPS_MAX}

{RE_DUP_MAX}
Maximum number of repeated occurrences of a regular expression permitted when using the interval notation \{*m*,*n*\}; see Chapter 9 (on page 165).
Minimum Acceptable Value: {_POSIX2_RE_DUP_MAX}

Maximum Values

TMR The symbolic constants in the following list shall be defined in **<limits.h>** with the values shown. These are symbolic names for the most restrictive value for certain features on an implementation supporting the Timers option. A conforming implementation shall provide values no larger than these values. A conforming application must not require a smaller value for correct operation.

TMR {_POSIX_CLOCKRES_MIN}
The resolution of the CLOCK_REALTIME clock, in nanoseconds.
Value: 20 000 000

MON If the Monotonic Clock option is supported, the resolution of the CLOCK_MONOTONIC clock, in nanoseconds, is represented by {_POSIX_CLOCKRES_MIN}.

Minimum Values

The symbolic constants in the following list shall be defined in **<limits.h>** with the values shown. These are symbolic names for the most restrictive value for certain features on an implementation conforming to this volume of IEEE Std 1003.1-200x. Related symbolic constants are defined elsewhere in this volume of IEEE Std 1003.1-200x which reflect the actual implementation and which need not be as restrictive. A conforming implementation shall provide values at least this large. A strictly conforming application must not require a larger value for correct operation.

AIO {_POSIX_AIO_LISTIO_MAX}
The number of I/O operations that can be specified in a list I/O call.
Value: 2

AIO {_POSIX_AIO_MAX}
The number of outstanding asynchronous I/O operations.
Value: 1

{_POSIX_ARG_MAX}
Maximum length of argument to the *exec* functions including environment data.
Value: 4 096

{_POSIX_CHILD_MAX}
Maximum number of simultaneous processes per real user ID.
Value: 6

TMR {_POSIX_DELAYTIMER_MAX}
The number of timer expiration overruns.
Value: 32

{_POSIX_HOST_NAME_MAX}
Maximum length of a host name (not including the terminating null) as returned from the *gethostname*( ) function.
Value: 255

{_POSIX_LINK_MAX}
Maximum number of links to a single file.
Value: 8

{_POSIX_LOGIN_NAME_MAX}
The size of the storage required for a login name, in bytes, including the terminating null.
Value: 9

{_POSIX_MAX_CANON}
Maximum number of bytes in a terminal canonical input queue.
Value: 255

{_POSIX_MAX_INPUT}
Maximum number of bytes allowed in a terminal input queue.
Value: 255

MSG {_POSIX_MQ_OPEN_MAX}
The number of message queues that can be open for a single process.
Value: 8

MSG {_POSIX_MQ_PRIO_MAX}
The maximum number of message priorities supported by the implementation.
Value: 32

{_POSIX_NAME_MAX}
Maximum number of bytes in a filename (not including terminating null).
Value: 14

{_POSIX_NGROUPS_MAX}
Maximum number of simultaneous supplementary group IDs per process.
Value: 8

{_POSIX_OPEN_MAX}
Maximum number of files that one process can have open at any one time.
Value: 20

{_POSIX_PATH_MAX}
Maximum number of bytes in a pathname.
Value: 256

{_POSIX_PIPE_BUF}
Maximum number of bytes that is guaranteed to be atomic when writing to a pipe.
Value: 512

{_POSIX_RE_DUP_MAX}
The number of repeated occurrences of a BRE permitted by the *regexec*() and *regcomp*() functions when using the interval notation {\(*m*,*n*\}; see Section 9.3.6 (on page 170).
Value: 255

RTS {_POSIX_RTSIG_MAX}
The number of realtime signal numbers reserved for application use.
Value: 8

SEM {_POSIX_SEM_NSEMS_MAX}
The number of semaphores that a process may have.
Value: 256

SEM {_POSIX_SEM_VALUE_MAX}
The maximum value a semaphore may have.
Value: 32 767

RTS {_POSIX_SIGQUEUE_MAX}
The number of queued signals that a process may send and have pending at the receiver(s) at any time.
Value: 32

{_POSIX_SSIZE_MAX}
The value that can be stored in an object of type **ssize_t**.
Value: 32 767

{_POSIX_STREAM_MAX}
The number of streams that one process can have open at one time.
Value: 8

SS|TSP {_POSIX_SS_REPL_MAX}
The number of replenishment operations that may be simultaneously pending for a particular sporadic server scheduler.
Value: 4

{_POSIX_SYMLINK_MAX}
The number of bytes in a symbolic link.
Value: 255

{_POSIX_SYMLOOP_MAX}
The number of symbolic links that can be traversed in the resolution of a pathname in the absence of a loop. |
Value: 8

THR {_POSIX_THREAD_DESTRUCTOR_ITERATIONS}
The number of attempts made to destroy a thread's thread-specific data values on thread exit.
Value: 4

THR {_POSIX_THREAD_KEYS_MAX}
The number of data keys per process.
Value: 128

THR {_POSIX_THREAD_THREADS_MAX}
The number of threads per process.
Value: 64

TMR {_POSIX_TIMER_MAX}
The per process number of timers.
Value: 32

TRC {_POSIX_TRACE_EVENT_NAME_MAX}
The length in bytes of a trace event name.
Value: 30

TRC {_POSIX_TRACE_NAME_MAX}
The length in bytes of a trace generation version string or a trace stream name.
Value: 8

TRC {_POSIX_TRACE_SYS_MAX}
The number of trace streams that may simultaneously exist in the system.
Value: 8

TRC {_POSIX_TRACE_USER_EVENT_MAX}
The number of user trace event type identifiers that may simultaneously exist in a traced process, including the predefined user trace event POSIX_TRACE_UNNAMED_USER_EVENT.
Value: 32

{_POSIX_TTY_NAME_MAX}
The size of the storage required for a terminal device name, in bytes, including the

terminating null.
Value: 9

{_POSIX_TZNAME_MAX}
Maximum number of bytes supported for the name of a timezone (not of the TZ variable).
Value: 6

Note: The length given by {_POSIX_TZNAME_MAX} does not include the quoting characters mentioned in Section 8.3 (on page 161).

{_POSIX2_BC_BASE_MAX}
Maximum *obase* values allowed by the *bc* utility.
Value: 99

{_POSIX2_BC_DIM_MAX}
Maximum number of elements permitted in an array by the *bc* utility.
Value: 2 048

{_POSIX2_BC_SCALE_MAX}
Maximum *scale* value allowed by the *bc* utility.
Value: 99

{_POSIX2_BC_STRING_MAX}
Maximum length of a string constant accepted by the *bc* utility.
Value: 1 000

{_POSIX2_CHARCLASS_NAME_MAX}
Maximum number of bytes in a character class name.
Value: 14

{_POSIX2_COLL_WEIGHTS_MAX}
Maximum number of weights that can be assigned to an entry of the *LC_COLLATE* **order** keyword in the locale definition file; see Chapter 7 (on page 119).
Value: 2

{_POSIX2_EXPR_NEST_MAX}
Maximum number of expressions that can be nested within parentheses by the *expr* utility.
Value: 32

{_POSIX2_LINE_MAX}
Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline.
Value: 2 048

{_POSIX2_RE_DUP_MAX]
Maximum number of repeated occurrences of a regular expression permitted when using the interval notation \{*m*,*n*\}; see Chapter 9 (on page 165).
Value: 255

XSI {_XOPEN_IOV_MAX}
Maximum number of **iovec** structures that one process has available for use with *readv*( ) or *writev*( ).
Value: 16

XSI {_XOPEN_NAME_MAX}
Maximum number of bytes in a filename (not including terminating null).
Value: 255

XSI {_XOPEN_PATH_MAX}
Maximum number of bytes in a pathname.
Value: 1 024

Numerical Limits

The values in the following lists shall be defined in **<limits.h>** and are constant expressions suitable for use in **#if** preprocessing directives. Moreover, except for {CHAR_BIT}, XSI {DBL_DIG}, {DBL_MAX}, {FLT_DIG}, {FLT_MAX}, {LONG_BIT}, {WORD_BIT}, and {MB_LEN_MAX}, the symbolic names are defined as expressions of the correct type.

If the value of an object of type **char** is treated as a signed integer when used in an expression, the value of {CHAR_MIN} is the same as that of {SCHAR_MIN} and the value of {CHAR_MAX} is the same as that of {SCHAR_MAX}. Otherwise, the value of {CHAR_MIN} is 0 and the value of {CHAR_MAX} is the same as that of {UCHAR_MAX}.

{CHAR_BIT}
Number of bits in a type **char**.
CX Value: 8

{CHAR_MAX}
Maximum value of type **char**.
Minimum Acceptable Value: {UCHAR_MAX} or {SCHAR_MAX}

{INT_MAX}
Maximum value of an **int**.
Minimum Acceptable Value: 2 147 483 647

XSI {LONG_BIT}
Number of bits in a **long**.
Minimum Acceptable Value: 32

{LONG_MAX}
Maximum value of a **long**.
Minimum Acceptable Value: +2 147 483 647

{MB_LEN_MAX}
Maximum number of bytes in a character, for any supported locale.
Minimum Acceptable Value: 1

{SCHAR_MAX}
Maximum value of type **signed char**.
CX Value: +127

{SHRT_MAX}
Maximum value of type **short**.
Minimum Acceptable Value: +32 767

{SSIZE_MAX}
Maximum value of an object of type **ssize_t**.
Minimum Acceptable Value: {_POSIX_SSIZE_MAX}

{UCHAR_MAX}
Maximum value of type **unsigned char**.
CX Value: 255

{UINT_MAX}
Maximum value of type **unsigned**.
Minimum Acceptable Value: 4 294 967 295

{ULONG_MAX}
Maximum value of type **unsigned long**.
Minimum Acceptable Value: 4 294 967 295

{USHRT_MAX}
Maximum value for a type **unsigned short**.
Minimum Acceptable Value: 65 535

XSI {WORD_BIT}
Number of bits in a word or type **int**.
Minimum Acceptable Value: 16

{CHAR_MIN}
Minimum value of type **char**.
Maximum Acceptable Value: {SCHAR_MIN} or 0

{INT_MIN}
Minimum value of type **int**.
Maximum Acceptable Value: –2 147 483 647

{LONG_MIN}
Minimum value of type **long**.
Maximum Acceptable Value: –2 147 483 647

{SCHAR_MIN}
Minimum value of type **signed char**.
CX Value: –128 |

{SHRT_MIN}
Minimum value of type **short**.
Maximum Acceptable Value: –32 767

{LLONG_MIN}
Minimum value of type **long long**.
Maximum Acceptable Value: –9223372036854775807

{LLONG_MAX}
Maximum value of type **long long**.
Minimum Acceptable Value: +9223372036854775807

{ULLONG_MAX}
Maximum value of type **unsigned long long**.
Minimum Acceptable Value: 18446744073709551615

Other Invariant Values

XSI The following constants shall be defined on all implementations in **<limits.h>**:

XSI {CHARCLASS_NAME_MAX}
Maximum number of bytes in a character class name.
Minimum Acceptable Value: 14

XSI {NL_ARGMAX}
Maximum value of *digit* in calls to the *printf*( ) and *scanf*( ) functions.
Minimum Acceptable Value: 9

XSI {NL_LANGMAX}
Maximum number of bytes in a *LANG* name.
Minimum Acceptable Value: 14

XSI {NL_MSGMAX}
Maximum message number.
Minimum Acceptable Value: 32 767

XSI {NL_NMAX}
Maximum number of bytes in an N-to-1 collation mapping.
Minimum Acceptable Value: '*'

XSI {NL_SETMAX}
Maximum set number.
Minimum Acceptable Value: 255

XSI {NL_TEXTMAX}
Maximum number of bytes in a message string.
Minimum Acceptable Value: {_POSIX2_LINE_MAX}

XSI {NZERO}
Default process priority.
Minimum Acceptable Value: 20

APPLICATION USAGE

None.

RATIONALE

A request was made to reduce the value of {_POSIX_LINK_MAX} from the value of 8 specified for it in the POSIX.1-1990 standard to 2. The standard developers decided to deny this request for several reasons.

- They wanted to avoid making any changes to the standard that could break conforming applications, and the requested change could have that effect.
- The use of multiple hard links to a file cannot always be replaced with use of symbolic links. Symbolic links are semantically different from hard links in that they associate a pathname with another pathname rather than a pathname with a file. This has implications for access control, file permanence, and transparency. |
- The original standard developers had considered the issue of allowing for implementations that did not in general support hard links, and decided that this would reduce consensus on the standard.

Systems that support historical versions of the development option of the ISO POSIX-2 standard retain the name {_POSIX2_RE_DUP_MAX} as an alias for {_POSIX_RE_DUP_MAX}.

{PATH_MAX}
IEEE PASC Interpretation 1003.1 #15 addressed the inconsistency in the standard with the definition of pathname and the description of {PATH_MAX}, allowing application writers to allocate either {PATH_MAX} or {PATH_MAX}+1 bytes. The inconsistency has been removed by correction to the {PATH_MAX} definition to include the null character. With this change, applications that previously allocated {PATH_MAX} bytes will continue to succeed. |

{SYMLINK_MAX}
This symbol refers to space for data that is stored in the file system, as opposed to {PATH_MAX} which is the length of a name that can be passed to a function. In some existing implementations, the filenames pointed to by symbolic links are stored in the inodes of the links, so it is important that {SYMLINK_MAX} not be constrained to be as large as {PATH_MAX}.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *fpathconf*( ), *pathconf*( ), *sysconf*( )

CHANGE HISTORY

First released in Issue 1.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

{FILESIZEBITS} added for the Large File Summit extensions.

The minimum acceptable values for {INT_MAX}, {INT_MIN}, and {UINT_MAX} are changed to make 32-bit values the minimum requirement.

The entry is restructured to improve readability.

Issue 6

The Open Group Corrigendum U033/4 is applied. The wording is made clear for {CHAR_MIN}, {INT_MIN}, {LONG_MIN}, {SCHAR_MIN}, and {SHRT_MIN} that these are maximum acceptable values.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The minimum value for {CHILD_MAX} is 25. This is a FIPS requirement.
- The minimum value for {OPEN_MAX} is 20. This is a FIPS requirement.
- The minimum value for {NGROUPS_MAX} is 8. This is also a FIPS requirement.

Symbolic constants are added for {_POSIX_SYMLINK_MAX}, {_POSIX_SYMLOOP_MAX}, {_POSIX_RE_DUP_MAX}, {RE_DUP_MAX}, {SYMLOOP_MAX}, and {SYMLINK_MAX}.

The following values are added for alignment with IEEE Std 1003.1d-1999:

{_POSIX_SS_REPL_MAX}
{SS_REPL_MAX}
{POSIX_ALLOC_SIZE_MIN}
{POSIX_REC_INCR_XFER_SIZE}
{POSIX_REC_MAX_XFER_SIZE}
{POSIX_REC_MIN_XFER_SIZE}
{POSIX_REC_XFER_ALIGN}

Reference to CLOCK_MONOTONIC is added in the description of {_POSIX_CLOCKRES_MIN} for alignment with IEEE Std 1003.1j-2000.

The constants {LLONG_MIN}, {LLONG_MAX}, and {ULLONG_MAX} are added for alignment with the ISO/IEC 9899:1999 standard.

The following values are added for alignment with IEEE Std 1003.1q-2000: |

{_POSIX_TRACE_EVENT_NAME_MAX}
{_POSIX_TRACE_NAME_MAX}
{_POSIX_TRACE_SYS_MAX}
{_POSIX_TRACE_USER_EVENT_MAX}
{TRACE_EVENT_NAME_MAX}
{TRACE_NAME_MAX}
{TRACE_SYS_MAX}
{TRACE_USER_EVENT_MAX}

The new limits {_XOPEN_NAME_MAX} and {_XOPEN_PATH_MAX} are added as minimum values for {PATH_MAX} and {NAME_MAX} limits on XSI-conformant systems.

The legacy symbols {PASS_MAX} and {TMP_MAX} are removed.

The values for the limits {CHAR_BIT}, {CHAR_MAX}, {SCHAR_MAX}, and {UCHAR_MAX} are now required to be 8, +127, 255, and −128, respectively.

NAME

locale.h — category macros

SYNOPSIS

```
#include <locale.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<locale.h>** header shall provide a definition for structure **lconv**, which shall include at least the following members. (See the definitions of *LC_MONETARY* in the Section 7.3.3 (on page 138), and Section 7.4.4 (on page 141).)

```
char    *currency_symbol
char    *decimal_point
char     frac_digits
char    *grouping
char    *int_curr_symbol
char     int_frac_digits
char     int_n_cs_precedes
char     int_n_sep_by_space
char     int_n_sign_posn
char     int_p_cs_precedes
char     int_p_sep_by_space
char     int_p_sign_posn
char    *mon_decimal_point
char    *mon_grouping
char    *mon_thousands_sep
char    *negative_sign
char     n_cs_precedes
char     n_sep_by_space
char     n_sign_posn
char    *positive_sign
char     p_cs_precedes
char     p_sep_by_space
char     p_sign_posn
char    *thousands_sep
```

The **<locale.h>** header shall define NULL (as defined in **<stddef.h>**) and at least the following as macros:

LC_ALL
LC_COLLATE
LC_CTYPE
CX *LC_MESSAGES*
LC_MONETARY
LC_NUMERIC
LC_TIME

which shall expand to distinct integer constant expressions, for use as the first argument to the *setlocale*( ) function.

Additional macro definitions, beginning with the characters *LC_* and an uppercase letter, may also be given here.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
struct  lconv *localeconv (void);
char   *setlocale(int, const char *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *localeconv*( ), *setlocale*( ), Chapter 8 (on page 157)

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ISO C standard.

Issue 6

The **lconv** structure is expanded with new members (**int_n_cs_precedes**, **int_n_sep_by_space**, **int_n_sign_posn**, **int_p_cs_precedes**, **int_p_sep_by_space**, and **int_p_sign_posn**) for alignment with the ISO / IEC 9899: 1999 standard.

Extensions beyond the ISO C standard are now marked.

NAME

math.h — mathematical declarations

SYNOPSIS

```
#include <math.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<math.h>** header shall include definitions for at least the following types:

float_t	A real-floating type at least as wide as **float**.
double_t	A real-floating type at least as wide as **double**, and at least as wide as **float_t**.

If FLT_EVAL_METHOD equals 0, **float_t** and **double_t** shall be **float** and **double**, respectively; if FLT_EVAL_METHOD equals 1, they shall both be **double**; if FLT_EVAL_METHOD equals 2, they shall both be **long double**; for other values of FLT_EVAL_METHOD, they are otherwise implementation-defined.

The **<math.h>** header shall define the following macros, where **real-floating** indicates that the argument shall be an expression of real-floating type:

```
int fpclassify(real-floating x);
int isfinite(real-floating x);
int isinf(real-floating x);
int isnan(real-floating x);
int isnormal(real-floating x);
int signbit(real-floating x);
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
```

The **<math.h>** header shall provide for the following constants. The values are of type **double** and are accurate within the precision of the **double** type.

XSI

M_E	Value of e
M_LOG2E	Value of $\log_2 e$
M_LOG10E	Value of $\log_{10} e$
M_LN2	Value of $\log_e 2$
M_LN10	Value of $\log_e 10$
M_PI	Value of π
M_PI_2	Value of $\pi/2$
M_PI_4	Value of $\pi/4$
M_1_PI	Value of $1/\pi$
M_2_PI	Value of $2/\pi$

M_2_SQRTPI	Value of $2/\sqrt{\pi}$
M_SQRT2	Value of $\sqrt{2}$
M_SQRT1_2	Value of $1/\sqrt{2}$

The header shall define the following symbolic constants:

XSI MAXFLOAT	Value of maximum non-infinite single-precision floating-point number.
HUGE_VAL	A positive **double** expression, not necessarily representable as a **float**. Used as an error value returned by the mathematics library. HUGE_VAL evaluates to +infinity on systems supporting IEEE Std 754-1985.
HUGE_VALF	A positive **float** constant expression. Used as an error value returned by the mathematics library. HUGE_VALF evaluates to +infinity on systems supporting IEEE Std 754-1985.
HUGE_VALL	A positive **long double** constant expression. Used as an error value returned by the mathematics library. HUGE_VALL evaluates to +infinity on systems supporting IEEE Std 754-1985.
INFINITY	A constant expression of type **float** representing positive or unsigned infinity, if available; else a positive constant of type **float** that overflows at translation time.
NAN	A constant expression of type **float** representing a quiet NaN. This symbolic constant is only defined if the implementation supports quiet NaNs for the **float** type.

The following macros shall be defined for number classification. They represent the mutually-exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with FP_ and an uppercase letter, may also be specified by the implementation.

FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO

The following optional macros indicate whether the *fma*( ) family of functions are fast compared with direct code:

FP_FAST_FMA
FP_FAST_FMAF
FP_FAST_FMAL

The FP_FAST_FMA macro shall be defined to indicate that the *fma*( ) function generally executes about as fast as, or faster than, a multiply and an add of **double** operands. The other macros have the equivalent meaning for the **float** and **long double** versions.

The following macros shall expand to integer constant expressions whose values are returned by *ilogb*(*x*) if *x* is zero or NaN, respectively. The value of FP_ILOGB0 shall be either {INT_MIN} or −{INT_MAX}. The value of FP_ILOGBNAN shall be either {INT_MAX} or {INT_MIN}.

FP_ILOGB0
FP_ILOGBNAN

The following macros shall expand to the integer constants 1 and 2, respectively;

MATH_ERRNO
MATH_ERREXCEPT

The following macro shall expand to an expression that has type **int** and the value MATH_ERRNO, MATH_ERREXCEPT, or the bitwise-inclusive OR of both:

math_errhandling

The value of math_errhandling is constant for the duration of the program. It is unspecified whether math_errhandling is a macro or an identifier with external linkage. If a macro definition is suppressed or a program defines an identifier with the name math_errhandling , the behavior is undefined. If the expression (math_errhandling & MATH_ERREXCEPT) can be non-zero, the implementation shall define the macros FE_DIVBYZERO, FE_INVALID, and FE_OVERFLOW in **<fenv.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
double      acos(double);
float       acosf(float);
double      acosh(double);
float       acoshf(float);
long double acoshl(long double);
long double acosl(long double);
double      asin(double);
float       asinf(float);
double      asinh(double);
float       asinhf(float);
long double asinhl(long double);
long double asinl(long double);
double      atan(double);
double      atan2(double, double);
float       atan2f(float, float);
long double atan2l(long double, long double);
float       atanf(float);
double      atanh(double);
float       atanhf(float);
long double atanhl(long double);
long double atanl(long double);
double      cbrt(double);
float       cbrtf(float);
long double cbrtl(long double);
double      ceil(double);
float       ceilf(float);
long double ceill(long double);
double      copysign(double, double);
float       copysignf(float, float);
long double copysignl(long double, long double);
double      cos(double);
float       cosf(float);
double      cosh(double);
float       coshf(float);
long double coshl(long double);
```

```
        long double cosl(long double);
        double      erf(double);
        double      erfc(double);
        float       erfcf(float);
        long double erfcl(long double);
        float       erff(float);
        long double erfl(long double);
        double      exp(double);
        double      exp2(double);
        float       exp2f(float);
        long double exp2l(long double);
        float       expf(float);
        long double expl(long double);
        double      expm1(double);
        float       expm1f(float);
        long double expm1l(long double);
        double      fabs(double);
        float       fabsf(float);
        long double fabsl(long double);
        double      fdim(double, double);
        float       fdimf(float, float);
        long double fdiml(long double, long double);
        double      floor(double);
        float       floorf(float);
        long double floorl(long double);
        double      fma(double, double, double);
        float       fmaf(float, float, float);
        long double fmal(long double, long double, long double);
        double      fmax(double, double);
        float       fmaxf(float, float);
        long double fmaxl(long double, long double);
        double      fmin(double, double);
        float       fminf(float, float);
        long double fminl(long double, long double);
        double      fmod(double, double);
        float       fmodf(float, float);
        long double fmodl(long double, long double);
        double      frexp(double, int *);
        float       frexpf(float value, int *);
        long double frexpl(long double value, int *);
        double      hypot(double, double);
        float       hypotf(float, float);
        long double hypotl(long double, long double);
        int         ilogb(double);
        int         ilogbf(float);
        int         ilogbl(long double);
XSI     double      j0(double);
        double      j1(double);
        double      jn(int, double);
        double      ldexp(double, int);
        float       ldexpf(float, int);
        long double ldexpl(long double, int);
```

```
double      lgamma(double);
float       lgammaf(float);
long double lgammal(long double);
long long   llrint(double);
long long   llrintf(float);
long long   llrintl(long double);
long long   llround(double);
long long   llroundf(float);
long long   llroundl(long double);
double      log(double);
double      log10(double);
float       log10f(float);
long double log10l(long double);
double      log1p(double);
float       log1pf(float);
long double log1pl(long double);
double      log2(double);
float       log2f(float);
long double log2l(long double);
double      logb(double);
float       logbf(float);
long double logbl(long double);
float       logf(float);
long double logl(long double);
long        lrint(double);
long        lrintf(float);
long        lrintl(long double);
long        lround(double);
long        lroundf(float);
long        lroundl(long double);
double      modf(double, double *);
float       modff(float, float *);
long double modfl(long double, long double *);
double      nan(const char *);
float       nanf(const char *);
long double nanl(const char *);
double      nearbyint(double);
float       nearbyintf(float);
long double nearbyintl(long double);
double      nextafter(double, double);
float       nextafterf(float, float);
long double nextafterl(long double, long double);
double      nexttoward(double, long double);
float       nexttowardf(float, long double);
long double nexttowardl(long double, long double);
double      pow(double, double);
float       powf(float, float);
long double powl(long double, long double);
double      remainder(double, double);
float       remainderf(float, float);
long double remainderl(long double, long double);
double      remquo(double, double, int *);
```

```
float       remquof(float, float, int *);
long double remquol(long double, long double, int *);
double      rint(double);
float       rintf(float);
long double rintl(long double);
double      round(double);
float       roundf(float);
long double roundl(long double);
```

XSI

```
double      scalb(double, double);
```

```
double      scalbln(double, long);
float       scalblnf(float, long);
long double scalblnl(long double, long);
double      scalbn(double, int);
float       scalbnf(float, int);
long double scalbnl(long double, int);
double      sin(double);
float       sinf(float);
double      sinh(double);
float       sinhf(float);
long double sinhl(long double);
long double sinl(long double);
double      sqrt(double);
float       sqrtf(float);
long double sqrtl(long double);
double      tan(double);
float       tanf(float);
double      tanh(double);
float       tanhf(float);
long double tanhl(long double);
long double tanl(long double);
double      tgamma(double);
float       tgammaf(float);
long double tgammal(long double);
double      trunc(double);
float       truncf(float);
long double truncl(long double);
```

XSI

```
double      y0(double);
double      y1(double);
double      yn(int, double);
```

The following external variable shall be defined:

XSI

```
extern int signgam;
```

The behavior of each of the functions defined in **<math.h>** is specified in the System Interfaces volume of IEEE Std 1003.1-200x for all representable values of its input arguments, except where stated otherwise. Each function shall execute as if it were a single operation without generating any externally visible exceptional conditions.

APPLICATION USAGE

The FP_CONTRACT pragma can be used to allow (if the state is on) or disallow (if the state is off) the implementation to contract expressions. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another FP_CONTRACT pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another FP_CONTRACT pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. The default state (on or off) for the pragma is implementation-defined.

RATIONALE

Before the ISO/IEC 9899:1999 standard, the math library was defined only for the floating type **double**. All the names formed by appending `'f'` or `'l'` to a name in **<math.h>** were reserved to allow for the definition of **float** and **long double** libraries; and the ISO/IEC 9899:1999 standard provides for all three versions of math functions.

The functions *ecvt*( ), *fcvt*( ), and *gcvt*( ) have been dropped from the ISO C standard since their capability is available through *sprintf*( ). These are provided on XSI-conformant systems supporting the Legacy Option Group.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *acos*( ), *acosh*( ), *asin*( ), *atan*( ), *atan2*( ), *cbrt*( ), *ceil*( ), *cos*( ), *cosh*( ), *erf*( ), *exp*( ), *expm1*( ), *fabs*( ), *floor*( ), *fmod*( ), *frexp*( ), *hypot*( ), *ilogb*( ), *isnan*( ), *j0*( ), *ldexp*( ), *lgamma*( ), *log*( ), *log10*( ), *log1p*( ), *logb*( ), *modf*( ), *nextafter*( ), *pow*( ), *remainder*( ), *rint*( ), *scalb*( ), *sin*( ), *sinh*( ), *sqrt*( ), *tan*( ), *tanh*( ), *y0*( )

CHANGE HISTORY

First released in Issue 1.

Issue 6

This reference page is updated to align with the ISO/IEC 9899:1999 standard.

NAME

monetary.h — monetary types

SYNOPSIS

XSI
```
#include <monetary.h>
```

DESCRIPTION

The **<monetary.h>** header shall define the following types:

size_t	As described in **<stddef.h>**.
ssize_t	As described in **<sys/types.h>**.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided.

```
ssize_t  strfmon(char *restrict, size_t, const char *restrict, ...);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *strfmon*( )

CHANGE HISTORY

First released in Issue 4.

Issue 6

The **restrict** keyword is added to the prototype for *strfmon*( ).

NAME

mqueue.h — message queues (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>
```

DESCRIPTION

The **<mqueue.h>** header shall define the **mqd_t** type, which is used for message queue descriptors. This is not an array type.

The **<mqueue.h>** header shall define the **sigevent** structure (as described in **<signal.h>**) and the **mq_attr** structure, which is used in getting and setting the attributes of a message queue. Attributes are initially set when the message queue is created. An **mq_attr** structure shall have at least the following fields:

`long`	`mq_flags`	Message queue flags.
`long`	`mq_maxmsg`	Maximum number of messages.
`long`	`mq_msgsize`	Maximum message size.
`long`	`mq_curmsgs`	Number of messages currently queued.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
int      mq_close(mqd_t);
int      mq_getattr(mqd_t, struct mq_attr *);
int      mq_notify(mqd_t, const struct sigevent *);
mqd_t    mq_open(const char *, int, ...);
ssize_t  mq_receive(mqd_t, char *, size_t, unsigned *);
int      mq_send(mqd_t, const char *, size_t, unsigned );
int      mq_setattr(mqd_t, const struct mq_attr *restrict,
             struct mq_attr *restrict);
```
TMO
```
ssize_t  mq_timedreceive(mqd_t, char *restrict, size_t,
             unsigned *restrict, const struct timespec *restrict);
int      mq_timedsend(mqd_t, const char *, size_t, unsigned ,
             const struct timespec *);
```
```
int      mq_unlink(const char *);
```

Inclusion of the **<mqueue.h>** header may make visible symbols defined in the headers **<fcntl.h>**, **<signal.h>**, **<sys/types.h>**, and **<time.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<fcntl.h>, **<signal.h>**, **<sys/types.h>**, **<time.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *mq_close*( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *mq_unlink*( )

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The **<mqueue.h>** header is marked as part of the Message Passing option.

The *mq_timedreceive*( ) and *mq_timedsend*( ) functions are added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the prototypes for *mq_setattr*( ) and *mq_timedreceive*( ).

<ndbm.h>

NAME

ndbm.h — definitions for ndbm database operations

SYNOPSIS

XSI

```
#include <ndbm.h>
```

DESCRIPTION

The **<ndbm.h>** header shall define the **datum** type as a structure that includes at least the following members:

`void`	`*dptr`	A pointer to the application's data.
`size_t`	`dsize`	The size of the object pointed to by *dptr*.

The **size_t** type shall be defined as described in **<stddef.h>**.

The **<ndbm.h>** header shall define the **DBM** type.

The following constants shall be defined as possible values for the *store_mode* argument to *dbm_store*( ):

DBM_INSERT	Insertion of new entries only.
DBM_REPLACE	Allow replacing existing entries.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int     dbm_clearerr(DBM *);
void    dbm_close(DBM *);
int     dbm_delete(DBM *, datum);
int     dbm_error(DBM *);
datum   dbm_fetch(DBM *, datum);
datum   dbm_firstkey(DBM *);
datum   dbm_nextkey(DBM *);
DBM    *dbm_open(const char *, int, mode_t);
int     dbm_store(DBM *, datum, datum, int);
```

The **mode_t** type shall be defined through **typedef** as described in **<sys/types.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *dbm_clearerr*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

References to the definitions of **size_t** and **mode_t** are added to the DESCRIPTION.

NAME

net / if.h — sockets local interfaces

SYNOPSIS

```
#include <net/if.h>
```

DESCRIPTION

The **<net/if.h>** header shall define the **if_nameindex** structure that includes at least the following members:

`unsigned  if_index`	Numeric index of the interface.
`char     *if_name`	Null-terminated name of the interface.

The **<net/if.h>** header shall define the following macro for the length of a buffer containing an interface name (including the terminating NULL character):

IF_NAMESIZE	Interface name length.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
unsigned              if_nametoindex(const char *);
char                 *if_indextoname(unsigned, char *);
struct if_nameindex  *if_nameindex(void);
void                  if_freenameindex(struct if_nameindex *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

netdb.h — definitions for network database operations

SYNOPSIS

```
#include <netdb.h>
```

DESCRIPTION

The **<netdb.h>** header may define the **in_port_t** type and the **in_addr_t** type as described in **<netinet/in.h>**.

The **<netdb.h>** header shall define the **hostent** structure that includes at least the following members:

`char`	`*h_name`	Official name of the host.
`char`	`**h_aliases`	A pointer to an array of pointers to alternative host names, terminated by a null pointer.
`int`	`h_addrtype`	Address type.
`int`	`h_length`	The length, in bytes, of the address.
`char`	`**h_addr_list`	A pointer to an array of pointers to network addresses (in network byte order) for the host, terminated by a null pointer.

The **<netdb.h>** header shall define the **netent** structure that includes at least the following members:

`char`	`*n_name`	Official, fully-qualified (including the domain) name of the host.
`char`	`**n_aliases`	A pointer to an array of pointers to alternative network names, terminated by a null pointer.
`int`	`n_addrtype`	The address type of the network.
`uint32_t`	`n_net`	The network number, in host byte order.

The **uint32_t** type shall be defined as described in **<inttypes.h>**.

The **<netdb.h>** header shall define the **protoent** structure that includes at least the following members:

`char`	`*p_name`	Official name of the protocol.
`char`	`**p_aliases`	A pointer to an array of pointers to alternative protocol names, terminated by a null pointer.
`int`	`p_proto`	The protocol number.

The **<netdb.h>** header shall define the **servent** structure that includes at least the following members:

`char`	`*s_name`	Official name of the service.
`char`	`**s_aliases`	A pointer to an array of pointers to alternative service names, terminated by a null pointer.
`int`	`s_port`	The port number at which the service resides, in network byte order.
`char`	`*s_proto`	The name of the protocol to use when contacting the service.

The **<netdb.h>** header shall define the IPPORT_RESERVED macro with the value of the highest reserved Internet port number.

OB When the **<netdb.h>** header is included, *h_errno* shall be available as a modifiable *l*-value of type **int**. It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage.

The **<netdb.h>** header shall define the following macros for use as error values for *gethostbyaddr*( ) and *gethostbyname*( ):

HOST_NOT_FOUND
NO_DATA
NO_RECOVERY
TRY_AGAIN

Address Information Structure

The **<netdb.h>** header shall define the **addrinfo** structure that includes at least the following members:

`int`	`ai_flags`	Input flags.
`int`	`ai_family`	Address family of socket.
`int`	`ai_socktype`	Socket type.
`int`	`ai_protocol`	Protocol of socket.
`socklen_t`	`ai_addrlen`	Length of socket address.
`struct sockaddr`	`*ai_addr`	Socket address of socket.
`char`	`*ai_canonname`	Canonical name of service location.
`struct addrinfo`	`*ai_next`	Pointer to next in list.

The **<netdb.h>** header shall define the following macros that evaluate to bitwise-distinct integer constants for use in the *flags* field of the **addrinfo** structure:

AI_PASSIVE	Socket address is intended for *bind*( ).
AI_CANONNAME	Request for canonical name.
AI_NUMERICHOST	Return numeric host address as name.
AI_NUMERICSERV	Inhibit service name resolution.
AI_V4MAPPED	If no IPv6 addresses are found, query for IPv4 addresses and return them to the caller as IPv4-mapped IPv6 addresses.
AI_ALL	Query for both IPv4 and IPv6 addresses.
AI_ADDRCONFIG	Query for IPv4 addresses only when an IPv4 address is configured; query for IPv6 addresses only when an IPv6 address is configured.

The **<netdb.h>** header shall define the following macros that evaluate to bitwise-distinct integer constants for use in the *flags* argument to *getnameinfo*( ):

NI_NOFQDN	Only the nodename portion of the FQDN is returned for local hosts.
NI_NUMERICHOST	The numeric form of the node's address is returned instead of its name.

NI_NAMEREQD	Return an error if the node's name cannot be located in the database.
NI_NUMERICSERV	The numeric form of the service address is returned instead of its name.
NI_DGRAM	Indicates that the service is a datagram service (SOCK_DGRAM).

Address Information Errors

The **<netdb.h>** header shall define the following macros for use as error values for *getaddrinfo*( ) and *getnameinfo*( ):

EAI_AGAIN	The name could not be resolved at this time. Future attempts may succeed.
EAI_BADFLAGS	The flags had an invalid value.
EAI_FAIL	A non-recoverable error occurred.
EAI_FAMILY	The address family was not recognized or the address length was invalid for the specified family.
EAI_MEMORY	There was a memory allocation failure.
EAI_NONAME	The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were null.
EAI_SERVICE	The service passed was not recognized for the specified socket type.
EAI_SOCKTYPE	The intended socket type was not recognized.
EAI_SYSTEM	A system error occurred. The error code can be found in *errno*.
EAI_OVERFLOW	An argument buffer overflowed.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
void             endhostent(void);
void             endnetent(void);
void             endprotoent(void);
void             endservent(void);
void             freeaddrinfo(struct addrinfo *);
const char      *gai_strerror(int);
int              getaddrinfo(const char *restrict, const char *restrict,
                     const struct addrinfo *restrict,
                     struct addrinfo **restrict);
struct hostent  *gethostbyaddr(const void *, socklen_t, int);
struct hostent  *gethostbyname(const char *);
struct hostent  *gethostent(void);
int              getnameinfo(const struct sockaddr *restrict, socklen_t,
                     char *restrict, socklen_t, char *restrict,
                     socklen_t, unsigned);
struct netent   *getnetbyaddr(uint32_t, int);
struct netent   *getnetbyname(const char *);
struct netent   *getnetent(void);
struct protoent *getprotobyname(const char *);
struct protoent *getprotobynumber(int);
struct protoent *getprotoent(void);
```

```
struct servent   *getservbyname(const char *, const char *);
struct servent   *getservbyport(int, const char *);
struct servent   *getservent(void);
void              sethostent(int);
void              setnetent(int);
void              setprotoent(int);
void              setservent(int);
```

The type **socklen_t** shall be defined through **typedef** as described in **<sys/socket.h>**.

Inclusion of the **<netdb.h>** header may also make visible all symbols from **<netinet/in.h>**, **<sys/socket.h>**, and **<inttypes.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<netinet/in.h>, **<inttypes.h>**, **<sys/socket.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *bind*( ), *endhostent*( ), *endnetent*( ), *endprotoent*( ), *endservent*( ), *getaddrinfo*( ), *getnameinfo*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The Open Group Base Resolution bwg2001-009 is applied, which changes the return type for *gai_strerror*( ) from **char *** to **const char ***. This is for coordination with the IPnG Working Group.

NAME

netinet/in.h — Internet address family

SYNOPSIS

```
#include <netinet/in.h>
```

DESCRIPTION

The **<netinet/in.h>** header shall define the following types:

in_port_t	Equivalent to the type **uint16_t** as defined in **<inttypes.h>**.
in_addr_t	Equivalent to the type **uint32_t** as defined in **<inttypes.h>**.

The **sa_family_t** type shall be defined as described in **<sys/socket.h>**.

The **uint8_t** and **uint32_t** type shall be defined as described in **<inttypes.h>**. Inclusion of the **<netinet/in.h>** header may also make visible all symbols from **<inttypes.h>** and **<sys/socket.h>**.

The **<netinet/in.h>** header shall define the **in_addr** structure that includes at least the following member:

```
in_addr_t  s_addr
```

The **<netinet/in.h>** header shall define the **sockaddr_in** structure that includes at least the following members (all in network byte order):

`sa_family_t`	`sin_family`	AF_INET.
`in_port_t`	`sin_port`	Port number.
`struct in_addr`	`sin_addr`	IP address.

The **sockaddr_in** structure is used to store addresses for the Internet address family. Values of this type shall be cast by applications to **struct sockaddr** for use with socket functions.

IP6 The **<netinet/in.h>** header shall define the **in6_addr** structure that contains at least the following member:

```
uint8_t s6_addr[16]
```

This array is used to contain a 128-bit IPv6 address, stored in network byte order.

The **<netinet/in.h>** header shall define the **sockaddr_in6** structure that includes at least the following members (all in network byte order):

`sa_family_t`	`sin6_family`	AF_INET6.
`in_port_t`	`sin6_port`	Port number.
`uint32_t`	`sin6_flowinfo`	IPv6 traffic class and flow information.
`struct in6_addr`	`sin6_addr`	IPv6 address.
`uint32_t`	`sin6_scope_id`	Set of interfaces for a scope.

The **sockaddr_in6** structure shall be set to zero by an application prior to using it, since implementations are free to have additional, implementation-defined fields in **sockaddr_in6**.

The *sin6_scope_id* field is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. For a link scope *sin6_addr*, *sin6_scope_id* would be an interface index. For a site scope *sin6_addr*, *sin6_scope_id* would be a site identifier. The mapping of *sin6_scope_id* to an interface or set of interfaces is implementation-defined.

The **<netinet/in.h>** header shall declare the following external variable:

```
struct in6_addr in6addr_any
```

This variable is initialized by the system to contain the wildcard IPv6 address. The **<netinet/in.h>** header also defines the IN6ADDR_ANY_INIT macro. This macro must be constant at compile time and can be used to initialize a variable of type **struct in6_addr** to the IPv6 wildcard address.

The **<netinet/in.h>** header shall declare the following external variable:

```
struct in6_addr in6addr_loopback
```

This variable is initialized by the system to contain the loopback IPv6 address. The **<netinet/in.h>** header also defines the IN6ADDR_LOOPBACK_INIT macro. This macro must be constant at compile time and can be used to initialize a variable of type **struct in6_addr** to the IPv6 loopback address.

The **<netinet/in.h>** header shall define the **ipv6_mreq** structure that includes at least the following members:

`struct in6_addr`	`ipv6mr_multiaddr`	IPv6 multicast address.
`unsigned`	`ipv6mr_interface`	Interface index.

The **<netinet/in.h>** header shall define the following macros for use as values of the *level* argument of *getsockopt*( ) and *setsockopt*( ):

	IPPROTO_IP	Internet protocol.
IP6	IPPROTO_IPV6	Internet Protocol Version 6.
	IPPROTO_ICMP	Control message protocol.
RS	IPPROTO_RAW	Raw IP Packets Protocol.
	IPPROTO_TCP	Transmission control protocol.
	IPPROTO_UDP	User datagram protocol.

The **<netinet/in.h>** header shall define the following macros for use as destination addresses for *connect*( ), *sendmsg*( ), and *sendto*( ):

INADDR_ANY	IPv4 local host address.
INADDR_BROADCAST	IPv4 broadcast address.

The **<netinet/in.h>** header shall define the following macro to help applications declare buffers of the proper size to store IPv4 addresses in string form:

INET_ADDRSTRLEN 16. Length of the string form for IP. |

The *htonl*( ), *htons*( ), *ntohl*( ), and *ntohs*( ) functions shall be available as defined in **<arpa/inet.h>**. Inclusion of the **<netinet/in.h>** header may also make visible all symbols from **<arpa/inet.h>**.

IP6 The **<netinet/in.h>** header shall define the following macro to help applications declare buffers of the proper size to store IPv6 addresses in string form:

INET6_ADDRSTRLEN 46. Length of the string form for IPv6. |

The **<netinet/in.h>** header shall define the following macros, with distinct integer values, for use in the *option_name* argument in the *getsockopt*( ) or *setsockopt*( ) functions at protocol level IPPROTO_IPV6:

IPV6_JOIN_GROUP	Join a multicast group.
IPV6_LEAVE_GROUP	Quit a multicast group.
IPV6_MULTICAST_HOPS	Multicast hop limit.

IPV6_MULTICAST_IF	Interface to use for outgoing multicast packets.
IPV6_MULTICAST_LOOP	Multicast packets are delivered back to the local application.
IPV6_UNICAST_HOPS	Unicast hop limit.
IPV6_V6ONLY	Restrict AF_INET6 socket to IPv6 communications only.

The **<netinet/in.h>** header shall define the following macros that test for special IPv6 addresses. Each macro is of type **int** and takes a single argument of type **const struct in6_addr ***:

IN6_IS_ADDR_UNSPECIFIED
Unspecified address.

IN6_IS_ADDR_LOOPBACK
Loopback address.

IN6_IS_ADDR_MULTICAST
Multicast address.

IN6_IS_ADDR_LINKLOCAL
Unicast link-local address.

IN6_IS_ADDR_SITELOCAL
Unicast site-local address.

IN6_IS_ADDR_V4MAPPED
IPv4 mapped address.

IN6_IS_ADDR_V4COMPAT
IPv4-compatible address.

IN6_IS_ADDR_MC_NODELOCAL
Multicast node-local address.

IN6_IS_ADDR_MC_LINKLOCAL
Multicast link-local address.

IN6_IS_ADDR_MC_SITELOCAL
Multicast site-local address.

IN6_IS_ADDR_MC_ORGLOCAL
Multicast organization-local address.

IN6_IS_ADDR_MC_GLOBAL
Multicast global address.

IN6_IS_ADDR_LINKLOCAL and IN6_IS_ADDR_SITELOCAL return true only for the two local-use IPv6 unicast addresses. They do not return true for multicast addresses of either link-local or site-local scope.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 4.8 (on page 97), **<arpa/inet.h>**, **<inttypes.h>**, **<sys/socket.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *connect*( ), *getsockopt*( ), *htonl*( ), *htons*( ), *ntohl*( ), *ntohs*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The *sin_zero* member was removed from the **sockaddr_in** structure as per The Open Group Base Resolution bwg2001-004.

NAME

netinet/tcp.h — definitions for the Internet Transmission Control Protocol (TCP)

SYNOPSIS

```
#include <netinet/tcp.h>
```

DESCRIPTION

The **<netinet/tcp.h>** header shall define the following macro for use as a socket option at the IPPROTO_TCP level:

TCP_NODELAY	Avoid coalescing of small segments.

The macro shall be defined in the header. The implementation need not allow the value of the option to be set via *setsockopt*( ) or retrieved via *getsockopt*( ).

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/socket.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *getsockopt*( ), *setsockopt*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

nl_types.h — data types

SYNOPSIS

XSI
```
#include <nl_types.h>
```

DESCRIPTION

The **<nl_types.h>** header shall contain definitions of at least the following types:

nl_catd	Used by the message catalog functions *catopen*( ), *catgets*( ), and *catclose*( ) to identify a catalog descriptor.
nl_item	Used by *nl_langinfo*( ) to identify items of *langinfo* data. Values of objects of type **nl_item** are defined in **<langinfo.h>**.

The **<nl_types.h>** header shall contain definitions of at least the following constants:

NL_SETD	Used by *gencat* when no $*set* directive is specified in a message text source file; see the Internationalization Guide. This constant can be passed as the value of *set_id* on subsequent calls to *catgets*( ) (that is, to retrieve messages from the default message set). The value of NL_SETD is implementation-defined.
NL_CAT_LOCALE	Value that must be passed as the *oflag* argument to *catopen*( ) to ensure that message catalog selection depends on the *LC_MESSAGES* locale category, rather than directly on the *LANG* environment variable.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int       catclose(nl_catd);
char     *catgets(nl_catd, int, int, const char *);
nl_catd   catopen(const char *, int);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<langinfo.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *catclose*( ), *catgets*( ), *catopen*( ), *nl_langinfo*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x, *gencat*

CHANGE HISTORY

First released in Issue 2.

NAME

poll.h — definitions for the poll( ) function

SYNOPSIS

XSI

```
#include <poll.h>
```

DESCRIPTION

The **<poll.h>** header shall define the **pollfd** structure that includes at least the following members:

`int`	`fd`	The following descriptor being polled.
`short`	`events`	The input event flags (see below).
`short`	`revents`	The output event flags (see below).

The **<poll.h>** header shall define the following type through **typedef**:

nfds_t	An unsigned integer type used for the number of file descriptors.

The implementation shall support one or more programming environments in which the width of **nfds_t** is no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

The following symbolic constants shall be defined, zero or more of which may be OR'ed together to form the *events* or *revents* members in the **pollfd** structure:

POLLIN	Data other than high-priority data may be read without blocking.
POLLRDNORM	Normal data may be read without blocking.
POLLRDBAND	Priority data may be read without blocking.
POLLPRI	High priority data may be read without blocking.
POLLOUT	Normal data may be written without blocking.
POLLWRNORM	Equivalent to POLLOUT.
POLLWRBAND	Priority data may be written.
POLLERR	An error has occurred (*revents* only).
POLLHUP	Device has been disconnected (*revents* only).
POLLNVAL	Invalid *fd* member (*revents* only).

The significance and semantics of normal, priority, and high-priority data are file and device-specific.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided.

```
int    poll(struct pollfd[], nfds_t, int);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *confstr*( ), *poll*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

The description of the symbolic constants is updated to match the *poll*( ) function.

Text related to STREAMS has been moved to the *poll*( ) reference page.

A note is added to the DESCRIPTION regarding the significance and semantics of normal, priority, and high-priority data.

NAME

pthread.h — threads

SYNOPSIS

THR
```
#include <pthread.h>
```

DESCRIPTION

The **<pthread.h>** header shall define the following symbols:

BAR PTHREAD_BARRIER_SERIAL_THREAD
PTHREAD_CANCEL_ASYNCHRONOUS
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_DISABLE
PTHREAD_CANCELED
PTHREAD_COND_INITIALIZER
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
PTHREAD_EXPLICIT_SCHED
PTHREAD_INHERIT_SCHED
XSI PTHREAD_MUTEX_DEFAULT
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_INITIALIZER
XSI PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_RECURSIVE
PTHREAD_ONCE_INIT
TPP | TPI PTHREAD_PRIO_INHERIT
PTHREAD_PRIO_NONE
PTHREAD_PRIO_PROTECT
PTHREAD_PROCESS_SHARED
PTHREAD_PROCESS_PRIVATE
TPS PTHREAD_SCOPE_PROCESS
PTHREAD_SCOPE_SYSTEM

The following types shall be defined as described in **<sys/types.h>**:

pthread_attr_t
BAR **pthread_barrier_t**
pthread_barrierattr_t
pthread_cond_t
pthread_condattr_t
pthread_key_t
pthread_mutex_t
pthread_mutexattr_t
pthread_once_t
pthread_rwlock_t
pthread_rwlockattr_t
SPI **pthread_spinlock_t**
pthread_t

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
          int    pthread_atfork(void (*)(void), void (*)(void),
                     void(*)(void));
          int    pthread_attr_destroy(pthread_attr_t *);
          int    pthread_attr_getdetachstate(const pthread_attr_t *, int *);
XSI       int    pthread_attr_getguardsize(const pthread_attr_t *restrict,
                     size_t *restrict);
TPS       int    pthread_attr_getinheritsched(const pthread_attr_t *restrict,
                     int *restrict);
          int    pthread_attr_getschedparam(const pthread_attr_t *restrict,
                     struct sched_param *restrict);
TPS       int    pthread_attr_getschedpolicy(const pthread_attr_t *restrict,
                     int *restrict);
TPS       int    pthread_attr_getscope(const pthread_attr_t *restrict,
                     int *restrict);
XSI       int    pthread_attr_getstack(const pthread_attr_t *restrict,
                     void **restrict, size_t *restrict);
TSA       int    pthread_attr_getstackaddr(const pthread_attr_t *restrict,
                     void **restrict);
          int    pthread_attr_getstacksize(const pthread_attr_t *restrict,
                     size_t *restrict);
          int    pthread_attr_init(pthread_attr_t *);
          int    pthread_attr_setdetachstate(pthread_attr_t *, int);
XSI       int    pthread_attr_setguardsize(pthread_attr_t *, size_t);
TPS       int    pthread_attr_setinheritsched(pthread_attr_t *, int);
          int    pthread_attr_setschedparam(pthread_attr_t *restrict,
                     const struct sched_param *restrict);
TPS       int    pthread_attr_setschedpolicy(pthread_attr_t *, int);
          int    pthread_attr_setscope(pthread_attr_t *, int);
XSI       int    pthread_attr_setstack(pthread_attr_t *, void *, size_t);
TSA       int    pthread_attr_setstackaddr(pthread_attr_t *, void *);
          int    pthread_attr_setstacksize(pthread_attr_t *, size_t);
BAR       int    pthread_barrier_destroy(pthread_barrier_t *);
          int    pthread_barrier_init(pthread_barrier_t *restrict,
                     const pthread_barrierattr_t *restrict, unsigned);
          int    pthread_barrier_wait(pthread_barrier_t *);
          int    pthread_barrierattr_destroy(pthread_barrierattr_t *);
          int    pthread_barrierattr_getpshared( \                           |
                     const pthread_barrierattr_t *restrict, int *restrict);  |
          int    pthread_barrierattr_init(pthread_barrierattr_t *);          |
          int    pthread_barrierattr_setpshared(pthread_barrierattr_t *, int);
          int    pthread_cancel(pthread_t);
          void   pthread_cleanup_push(void (*)(void *), void *);
          void   pthread_cleanup_pop(int);
          int    pthread_cond_broadcast(pthread_cond_t *);
          int    pthread_cond_destroy(pthread_cond_t *);
          int    pthread_cond_init(pthread_cond_t *restrict,
                     const pthread_condattr_t *restrict);
          int    pthread_cond_signal(pthread_cond_t *);
          int    pthread_cond_timedwait(pthread_cond_t *restrict,
                     pthread_mutex_t *restrict, const struct timespec *restrict);
          int    pthread_cond_wait(pthread_cond_t *restrict,
                     pthread_mutex_t *restrict);
```

```
          int   pthread_condattr_destroy(pthread_condattr_t *);
CS        int   pthread_condattr_getclock(const pthread_condattr_t *restrict,
                    clockid_t *restrict);
          int   pthread_condattr_getpshared(const pthread_condattr_t *restrict,
                    int *restrict);
          int   pthread_condattr_init(pthread_condattr_t *);
CS        int   pthread_condattr_setclock(pthread_condattr_t *, clockid_t);
          int   pthread_condattr_setpshared(pthread_condattr_t *, int);
          int   pthread_create(pthread_t *restrict, const pthread_attr_t *restrict,
                    void *(*)(void *), void *restrict);
          int   pthread_detach(pthread_t);
          int   pthread_equal(pthread_t, pthread_t);
          void  pthread_exit(void *);
XSI       int   pthread_getconcurrency(void);
TCT       int   pthread_getcpuclockid(pthread_t, clockid_t *);
TPS       int   pthread_getschedparam(pthread_t, int *restrict,
                    struct sched_param *restrict);
          void *pthread_getspecific(pthread_key_t);
          int   pthread_join(pthread_t, void **);
          int   pthread_key_create(pthread_key_t *, void (*)(void *));
          int   pthread_key_delete(pthread_key_t);
          int   pthread_mutex_destroy(pthread_mutex_t *);
TPP       int   pthread_mutex_getprioceiling(const pthread_mutex_t *restrict,
                    int *restrict);
          int   pthread_mutex_init(pthread_mutex_t *restrict,
                    const pthread_mutexattr_t *restrict);
          int   pthread_mutex_lock(pthread_mutex_t *);
TPP       int   pthread_mutex_setprioceiling(pthread_mutex_t *restrict, int,
                    int *restrict);
TMO       int   pthread_mutex_timedlock(pthread_mutex_t *,
                    const struct timespec *);
          int   pthread_mutex_trylock(pthread_mutex_t *);
          int   pthread_mutex_unlock(pthread_mutex_t *);
          int   pthread_mutexattr_destroy(pthread_mutexattr_t *);
TPP|TPI   int   pthread_mutexattr_getprioceiling( \                                |
                    const pthread_mutexattr_t *restrict, int *restrict);           |
          int   pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict, |
                    int *restrict);
          int   pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict,
                    int *restrict);
XSI       int   pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict,
                    int *restrict);
          int   pthread_mutexattr_init(pthread_mutexattr_t *);
TPP|TPI   int   pthread_mutexattr_setprioceiling(pthread_mutexattr_t *, int);
          int   pthread_mutexattr_setprotocol(pthread_mutexattr_t *, int);
          int   pthread_mutexattr_setpshared(pthread_mutexattr_t *, int);
XSI       int   pthread_mutexattr_settype(pthread_mutexattr_t *, int);
          int   pthread_once(pthread_once_t *, void (*)(void));
          int   pthread_rwlock_destroy(pthread_rwlock_t *);
          int   pthread_rwlock_init(pthread_rwlock_t *restrict,
                    const pthread_rwlockattr_t *restrict);
          int   pthread_rwlock_rdlock(pthread_rwlock_t *);
```

```
            int     pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict,
                        const struct timespec *restrict);
            int     pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict,
                        const struct timespec *restrict);
            int     pthread_rwlock_tryrdlock(pthread_rwlock_t *);
            int     pthread_rwlock_trywrlock(pthread_rwlock_t *);
            int     pthread_rwlock_unlock(pthread_rwlock_t *);
            int     pthread_rwlock_wrlock(pthread_rwlock_t *);
            int     pthread_rwlockattr_destroy(pthread_rwlockattr_t *);
            int     pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict,
                        int *restrict);
            int     pthread_rwlockattr_init(pthread_rwlockattr_t *);
            int     pthread_rwlockattr_setpshared(pthread_rwlockattr_t *, int);
            pthread_t
                    pthread_self(void);
            int     pthread_setcancelstate(int, int *);
            int     pthread_setcanceltype(int, int *);
XSI         int     pthread_setconcurrency(int);
TPS         int     pthread_setschedparam(pthread_t, int,
                        const struct sched_param *);
THR TPS     int     pthread_setschedprio(pthread_t, int);
            int     pthread_setspecific(pthread_key_t, const void *);
SPI         int     pthread_spin_destroy(pthread_spinlock_t *);
            int     pthread_spin_init(pthread_spinlock_t *, int);
            int     pthread_spin_lock(pthread_spinlock_t *);
            int     pthread_spin_trylock(pthread_spinlock_t *);
            int     pthread_spin_unlock(pthread_spinlock_t *);
            void    pthread_testcancel(void);
```

Inclusion of the **<pthread.h>** header shall make symbols defined in the headers **<sched.h>** and **<time.h>** visible.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sched.h>, **<time.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *pthread_attr_getguardsize*( ), *pthread_attr_init*( ), *pthread_attr_setscope*( ), *pthread_barrier_destroy*( ), *pthread_barrier_init*( ), *pthread_barrier_wait*( ), *pthread_barrierattr_destroy*( ), *pthread_barrierattr_getpshared*( ), *pthread_barrierattr_init*( ), *pthread_barrierattr_setpshared*( ), *pthread_cancel*( ), *pthread_cleanup_pop*( ), *pthread_cond_init*( ), *pthread_cond_signal*( ), *pthread_cond_wait*( ), *pthread_condattr_getclock*( ), *pthread_condattr_init*( ), *pthread_condattr_setclock*( ), *pthread_create*( ), *pthread_detach*( ), *pthread_equal*( ), *pthread_exit*( ), *pthread_getconcurrency*( ), *pthread_getcpuclockid*( ), *pthread_getschedparam*( ), *pthread_join*( ), *pthread_key_create*( ), *pthread_key_delete*( ), *pthread_mutex_init*( ), *pthread_mutex_lock*( ), *pthread_mutex_setprioceiling*( ), *pthread_mutex_timedlock*( ), *pthread_mutexattr_init*( ), *pthread_mutexattr_gettype*( ), *pthread_mutexattr_setprotocol*( ), *pthread_once*( ), *pthread_rwlock_destroy*( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), *pthread_rwlockattr_destroy*( ), *pthread_rwlockattr_getpshared*( ), *pthread_rwlockattr_init*( ), *pthread_rwlockattr_setpshared*( ), *pthread_self*( ), *pthread_setcancelstate*( ), *pthread_setspecific*( ), *pthread_spin_destroy*( ), *pthread_spin_init*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), *pthread_spin_unlock*( )

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The RTT margin markers are now broken out into their POSIX options.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the *pthread_cond_wait*( ) function.

The Open Group Corrigendum U026/2 is applied correcting the prototype for the *pthread_setschedparam*( ) function so that its second argument is of type **int**.

The *pthread_getcpuclockid*( ) and *pthread_mutex_timedlock*( ) functions are added for alignment with IEEE Std 1003.1d-1999.

The following functions are added for alignment with IEEE Std 1003.1j-2000: *pthread_barrier_destroy*( ), *pthread_barrier_init*( ), *pthread_barrier_wait*( ), *pthread_barrierattr_destroy*( ), *pthread_barrierattr_getpshared*( ), *pthread_barrierattr_init*( ), *pthread_barrierattr_setpshared*( ), *pthread_condattr_getclock*( ), *pthread_condattr_setclock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_spin_destroy*( ), *pthread_spin_init*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), and *pthread_spin_unlock*( ).

PTHREAD_RWLOCK_INITIALIZER is deleted for alignment with IEEE Std 1003.1j-2000.

Functions previously marked as part of the Read-Write Locks option are now moved to the Threads option.

The **restrict** keyword is added to the prototypes for *pthread_attr_getguardsize*( ), *pthread_attr_getinheritsched*( ), *pthread_attr_getschedparam*( ), *pthread_attr_getschedpolicy*( ), *pthread_attr_getscope*( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getstacksize*( ), *pthread_attr_setschedparam*( ), *pthread_barrier_init*( ), *pthread_barrierattr_getpshared*( ), *pthread_cond_init*( ), *pthread_cond_signal*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), *pthread_condattr_getclock*( ), *pthread_condattr_getpshared*( ), *pthread_create*( ), *pthread_getschedparam*( ), *pthread_mutex_getprioceiling*( ), *pthread_mutex_init*( ), *pthread_mutex_setprioceiling*( ), *pthread_mutexattr_getprioceiling*( ), *pthread_mutexattr_getprotocol*( ), *pthread_mutexattr_getpshared*( ), *pthread_mutexattr_gettype*( ), *pthread_rwlock_init*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlockattr_getpshared*( ), and *pthread_sigmask*( ).

IEEE PASC Interpretation 1003.1 #86 is applied, allowing the symbols from **<sched.h>** and **<time.h>** to be made visible when **<pthread.h>** is included. Previously this was an XSI extension.

IEEE PASC Interpretation 1003.1c #42 is applied, removing the requirement for prototypes for the *pthread_kill*( ) and *pthread_sigmask*( ) functions. These are required to be in the **<signal.h>** header. They are allowed here through the name space rules. |

IEEE PASC Interpretation 1003.1 #96 is applied, adding the *pthread_setschedprio*( ) function. |

NAME

pwd.h — password structure

SYNOPSIS

```
#include <pwd.h>
```

DESCRIPTION

The **<pwd.h>** header shall provide a definition for **struct passwd**, which shall include at least the following members:

`char`	`*pw_name`	User's login name.
`uid_t`	`pw_uid`	Numerical user ID.
`gid_t`	`pw_gid`	Numerical group ID.
`char`	`*pw_dir`	Initial working directory.
`char`	`*pw_shell`	Program to use as shell.

The **gid_t** and **uid_t** types shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
    struct passwd *getpwnam(const char *);
    struct passwd *getpwuid(uid_t);
TSF int            getpwnam_r(const char *, struct passwd *, char *,
                       size_t, struct passwd **);
    int            getpwuid_r(uid_t, struct passwd *, char *,
                       size_t, struct passwd **);
XSI void           endpwent(void);
    struct passwd *getpwent(void);
    void           setpwent(void);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *endpwent*( ), *getpwnam*( ), *getpwuid*( )

CHANGE HISTORY

First released in Issue 1.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **gid_t** and **uid_t** types are mandated.
- The *getpwnam_r*( ) and *getpwuid_r*( ) functions are marked as part of the _POSIX_THREAD_SAFE_FUNCTIONS option.

NAME

regex.h — regular expression matching types

SYNOPSIS

```
#include <regex.h>
```

DESCRIPTION

The **<regex.h>** header shall define the structures and symbolic constants used by the *regcomp*( ), *regexec*( ), *regerror*( ), and *regfree*( ) functions.

The structure type **regex_t** shall contain at least the following member:

`size_t`	`re_nsub`	Number of parenthesized subexpressions.

The type **size_t** shall be defined as described in **<sys/types.h>**.

The type **regoff_t** shall be defined as a signed integer type that can hold the largest value that can be stored in either a type **off_t** or type **ssize_t**. The structure type **regmatch_t** shall contain at least the following members:

`regoff_t`	`rm_so`	Byte offset from start of string to start of substring.
`regoff_t`	`rm_eo`	Byte offset from start of string of the first character after the end of substring.

Values for the *cflags* parameter to the *regcomp*( ) function:

REG_EXTENDED	Use Extended Regular Expressions.
REG_ICASE	Ignore case in match.
REG_NOSUB	Report only success or fail in *regexec*( ).
REG_NEWLINE	Change the handling of newline.

Values for the *eflags* parameter to the *regexec*( ) function:

REG_NOTBOL	The circumflex character (`'^'`), when taken as a special character, does not match the beginning of *string*.
REG_NOTEOL	The dollar sign (`'$'`), when taken as a special character, does not match the end of *string*.

The following constants shall be defined as error return values:

REG_NOMATCH	*regexec*( ) failed to match.
REG_BADPAT	Invalid regular expression.
REG_ECOLLATE	Invalid collating element referenced.
REG_ECTYPE	Invalid character class type referenced.
REG_EESCAPE	Trailing `'\'` in pattern.
REG_ESUBREG	Number in *digit* invalid or in error.
REG_EBRACK	`"[]"` imbalance.
REG_EPAREN	`"\(\)"` or `"()"` imbalance.
REG_EBRACE	`"\{\}"` imbalance.
REG_BADBR	Content of `"\{\}"` invalid: not a number, number too large, more than two numbers, first larger than second.

	REG_ERANGE	Invalid endpoint in range expression.
	REG_ESPACE	Out of memory.
	REG_BADRPT	`'?'`, `'*'`, or `'+'` not preceded by valid regular expression.
OB	REG_ENOSYS	Reserved.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    regcomp(regex_t *restrict, const char *restrict, int);
size_t regerror(int, const regex_t *restrict, char *restrict, size_t);
int    regexec(const regex_t *restrict, const char *restrict, size_t,
           regmatch_t[restrict], int);
void   regfree(regex_t *);
```

The implementation may define additional macros or constants using names beginning with REG_.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *regcomp*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4.

Originally derived from the ISO POSIX-2 standard.

Issue 6

The REG_ENOSYS constant is marked obsolescent.

The **restrict** keyword is added to the prototypes for *regcomp*( ), *regerror*( ), and *regexec*( ).

A statement is added that the **size_t** type is defined as described in **<sys/types.h>**.

NAME

sched.h — execution scheduling (**REALTIME**)

SYNOPSIS

PS
```
#include <sched.h>
```

DESCRIPTION

The **<sched.h>** header shall define the **sched_param** structure, which contains the scheduling parameters required for implementation of each supported scheduling policy. This structure shall contain at least the following member:

`int`	`sched_priority`	Process execution scheduling priority.

SS|TSP In addition, if _POSIX_SPORADIC_SERVER or _POSIX_THREAD_SPORADIC_SERVER is defined, the **sched_param** structure defined in **<sched.h>** shall contain the following members in addition to those specified above:

`int`	`sched_ss_low_priority`	Low scheduling priority for sporadic server.
`struct timespec`	`sched_ss_repl_period`	Replenishment period for sporadic server.
`struct timespec`	`sched_ss_init_budget`	Initial budget for sporadic server.
`int`	`sched_ss_max_repl`	Maximum pending replenishments for sporadic server.

Each process is controlled by an associated scheduling policy and priority. Associated with each policy is a priority range. Each policy definition specifies the minimum priority range for that policy. The priority ranges for each policy may overlap the priority ranges of other policies.

Four scheduling policies are defined; others may be defined by the implementation. The four standard policies are indicated by the values of the following symbolic constants:

	SCHED_FIFO	First in-first out (FIFO) scheduling policy.
	SCHED_RR	Round robin scheduling policy.
SS|TSP	SCHED_SPORADIC	Sporadic server scheduling policy.
	SCHED_OTHER	Another scheduling policy.

The values of these constants are distinct.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    sched_get_priority_max(int);
int    sched_get_priority_min(int);
int    sched_getparam(pid_t, struct sched_param *);
int    sched_getscheduler(pid_t);
int    sched_rr_get_interval(pid_t, struct timespec *);
int    sched_setparam(pid_t, const struct sched_param *);
int    sched_setscheduler(pid_t, int, const struct sched_param *);
int    sched_yield(void);
```

Inclusion of the **<sched.h>** header makes symbols defined in the header **<time.h>** visible.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<time.h>

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The **<sched.h>** header is marked as part of the Process Scheduling option.

Sporadic server members are added to the **sched_param** structure, and the SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #108 is applied, correcting the **sched_param** structure whose members *sched_ss_repl_period* and *sched_ss_init_budget* members should be type **struct timespec** and not **timespec**.

NAME

search.h — search tables

SYNOPSIS

XSI
```
#include <search.h>
```

DESCRIPTION

The **<search.h>** header shall define the **ENTRY** type for structure **entry** which shall include the following members:

```
char    *key
void    *data
```

and shall define **ACTION** and **VISIT** as enumeration data types through type definitions as follows:

```
enum { FIND, ENTER } ACTION;
enum { preorder, postorder, endorder, leaf } VISIT;
```

The **size_t** type shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    hcreate(size_t);
void   hdestroy(void);
ENTRY *hsearch(ENTRY, ACTION);
void   insque(void *, void *);
void  *lfind(const void *, const void *, size_t *,
          size_t, int (*)(const void *, const void *));
void  *lsearch(const void *, void *, size_t *,
          size_t, int (*)(const void *, const void *));
void   remque(void *);
void  *tdelete(const void *restrict, void **restrict,
          int(*)(const void *, const void *));
void  *tfind(const void *, void *const *,
          int(*)(const void *, const void *));
void  *tsearch(const void *, void **,
          int(*)(const void *, const void *));
void   twalk(const void *,
          void (*)(const void *, VISIT, int ));
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *hcreate*( ), *insque*( ), *lsearch*( ), *remque*( ), *tsearch*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The Open Group Corrigendum U021/6 is applied updating the prototypes for *tdelete*( ) and *tsearch*( ).

The **restrict** keyword is added to the prototype for *tdelete*( ).

<semaphore.h>

NAME

semaphore.h — semaphores (**REALTIME**)

SYNOPSIS

SEM
```
#include <semaphore.h>
```

DESCRIPTION

The **<semaphore.h>** header shall define the **sem_t** type, used in performing semaphore operations. The semaphore may be implemented using a file descriptor, in which case applications are able to open up at least a total of {OPEN_MAX} files and semaphores. The symbol SEM_FAILED shall be defined (see *sem_open*( )).

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
int    sem_close(sem_t *);
int    sem_destroy(sem_t *);
int    sem_getvalue(sem_t *restrict, int *restrict);
int    sem_init(sem_t *, int, unsigned);
sem_t *sem_open(const char *, int, ...);
int    sem_post(sem_t *);
int    sem_timedwait(sem_t *restrict, const struct timespec *restrict);
int    sem_trywait(sem_t *);
int    sem_unlink(const char *);
int    sem_wait(sem_t *);
```

TMO (sem_timedwait)

Inclusion of the **<semaphore.h>** header may make visible symbols defined in the headers **<fcntl.h>** and **<sys/types.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<fcntl.h>, **<sys/types.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( ), *sem_wait*( )

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The **<semaphore.h>** header is marked as part of the Semaphores option.

The Open Group Corrigendum U021/3 is applied, adding a description of SEM_FAILED.

The *sem_timedwait*( ) function is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the prototypes for *sem_getvalue*( ) and *sem_timedwait*( ).

NAME

setjmp.h — stack environment declarations

SYNOPSIS

```
#include <setjmp.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

CX The **<setjmp.h>** header shall define the array types **jmp_buf** and **sigjmp_buf**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
void   longjmp(jmp_buf, int);
void   siglongjmp(sigjmp_buf, int);
void  _longjmp(jmp_buf, int);
```

CX: siglongjmp; XSI: _longjmp

The following may be declared as a function, or defined as a macro, or both. Function prototypes shall be provided. |

```
int    setjmp(jmp_buf);
int    sigsetjmp(sigjmp_buf, int);
int   _setjmp(jmp_buf);
```

CX: sigsetjmp; XSI: _setjmp

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *longjmp*( ), *_longjmp*( ), *setjmp*( ), *siglongjmp*( ), *sigsetjmp*( )

CHANGE HISTORY

First released in Issue 1.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

signal.h — signals

SYNOPSIS

```
#include <signal.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<signal.h>** header shall define the following symbolic constants, each of which expands to a distinct constant expression of the type:

```
void (*)(int)
```

whose value matches no declarable function.

SIG_DFL	Request for default signal handling.
SIG_ERR	Return value from *signal*( ) in case of error.
CX SIG_HOLD	Request that signal be held.
SIG_IGN	Request that signal be ignored.

The following data types shall be defined through **typedef**:

sig_atomic_t	Possibly volatile-qualified integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
CX **sigset_t**	Integer or structure type of an object used to represent sets of signals.
CX **pid_t**	As described in **<sys/types.h>**.

RTS The **<signal.h>** header shall define the **sigevent** structure, which has at least the following members:

`int`	`sigev_notify`	Notification type.
`int`	`sigev_signo`	Signal number.
`union sigval`	`sigev_value`	Signal value.
`void(*)(union sigval)`	`sigev_notify_function`	Notification function.
`(pthread_attr_t *)`	`sigev_notify_attributes`	Notification attributes.

The following values of *sigev_notify* shall be defined:

SIGEV_NONE	No asynchronous notification is delivered when the event of interest occurs.
SIGEV_SIGNAL	A queued signal, with an application-defined value, is generated when the event of interest occurs.
SIGEV_THREAD	A notification function is called to perform notification.

The **sigval** union shall be defined as:

`int`	`sival_int`	Integer signal value.
`void`	`*sival_ptr`	Pointer signal value.

This header shall also declare the macros SIGRTMIN and SIGRTMAX, which evaluate to integer expressions, and specify a range of signal numbers that are reserved for application use and for which the realtime signal behavior specified in this volume of IEEE Std 1003.1-200x is supported.

The signal numbers in this range do not overlap any of the signals specified in the following table.

The range SIGRTMIN through SIGRTMAX inclusive shall include at least {RTSIG_MAX} signal numbers.

It is implementation-defined whether realtime signal behavior is supported for other signals.

This header also declares the constants that are used to refer to the signals that occur in the system. Signals defined here begin with the letters SIG. Each of the signals have distinct positive integer values. The value 0 is reserved for use as the null signal (see *kill*()). Additional implementation-defined signals may occur in the system.

CX The ISO C standard only requires the signal names SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM to be defined.

The following signals shall be supported on all implementations (default actions are explained below the table):

Signal	Default Action	Description
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped, or continued.
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.
SIGUSR2	T	User-defined signal 2.
SIGPOLL	T	Pollable event.
SIGPROF	T	Profiling timer expired.
SIGSYS	A	Bad system call.
SIGTRAP	A	Trace/breakpoint trap.
SIGURG	I	High bandwidth data is available at a socket.
SIGVTALRM	T	Virtual timer expired.
SIGXCPU	A	CPU time limit exceeded.
SIGXFSZ	A	File size limit exceeded.

The default actions are as follows:

T Abnormal termination of the process. The process is terminated with all the consequences of *_exit*() except that the status made available to *wait*() and *waitpid*() indicates abnormal termination by the specified signal.

A Abnormal termination of the process.

XSI Additionally, implementation-defined abnormal termination actions, such as creation of a core file, may occur.

I Ignore the signal.

S Stop the process.

C Continue the process, if it is stopped; otherwise, ignore the signal.

CX The header shall provide a declaration of **struct sigaction**, including at least the following members:

`void (*sa_handler)(int)`	What to do on receipt of signal.
`sigset_t sa_mask`	Set of signals to be blocked during execution of the signal handling function.
`int sa_flags`	Special flags.
`void (*)(int, siginfo_t *, void *) sa_sigaction`	Pointer to signal handler function or one of the macros SIG_IGN or SIG_DFL.

XSI The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a portable program must not use both simultaneously.

The following shall be declared as constants:

CX XSI	SA_NOCLDSTOP	Do not generate SIGCHLD when children stop or stopped children continue.
CX	SIG_BLOCK	The resulting set is the union of the current set and the signal set pointed to by the argument *set*.
CX	SIG_UNBLOCK	The resulting set is the intersection of the current set and the complement of the signal set pointed to by the argument *set*.
CX	SIG_SETMASK	The resulting set is the signal set pointed to by the argument *set*.
XSI	SA_ONSTACK	Causes signal delivery to occur on an alternate stack.
XSI	SA_RESETHAND	Causes signal dispositions to be set to SIG_DFL on entry to signal handlers.
XSI	SA_RESTART	Causes certain functions to become restartable.
XSI	SA_SIGINFO	Causes extra information to be passed to signal handlers at the time of receipt of a signal.
XSI	SA_NOCLDWAIT	Causes implementations not to create zombie processes on child death.
XSI	SA_NODEFER	Causes signal not to be automatically blocked on entry to signal handler.
XSI	SS_ONSTACK	Process is executing on an alternate signal stack.
XSI	SS_DISABLE	Alternate signal stack is disabled.
XSI	MINSIGSTKSZ	Minimum stack size for a signal handler.
XSI	SIGSTKSZ	Default size in bytes for the alternate signal stack.

XSI The **ucontext_t** structure shall be defined through **typedef** as described in **<ucontext.h>**.

The **mcontext_t** type shall be defined through **typedef** as described in **<ucontext.h>**.

The **<signal.h>** header shall define the **stack_t** type as a structure that includes at least the following members:

`void`	`*ss_sp`	Stack base or pointer.
`size_t`	`ss_size`	Stack size.
`int`	`ss_flags`	Flags.

The **<signal.h>** header shall define the **sigstack** structure that includes at least the following members:

`int`	`ss_onstack`	Non-zero when signal stack is in use.
`void`	`*ss_sp`	Signal stack pointer.

CX The **<signal.h>** header shall define the **siginfo_t** type as a structure that includes at least the following members:

CX	`int`	`si_signo`	Signal number.
XSI	`int`	`si_errno`	If non-zero, an *errno* value associated with this signal, as defined in **<errno.h>**.
CX	`int`	`si_code`	Signal code.
XSI	`pid_t`	`si_pid`	Sending process ID.
	`uid_t`	`si_uid`	Real user ID of sending process.
	`void`	`*si_addr`	Address of faulting instruction.
	`int`	`si_status`	Exit value or signal.
	`long`	`si_band`	Band event for SIGPOLL.
RTS	`union sigval`	`si_value`	Signal value.

The macros specified in the **Code** column of the following table are defined for use as values of XSI *si_code* that are signal-specific or non-signal-specific reasons why the signal was generated.

Signal	Code	Reason
XSI SIGILL	ILL_ILLOPC	Illegal opcode.
	ILL_ILLOPN	Illegal operand.
	ILL_ILLADR	Illegal addressing mode.
	ILL_ILLTRP	Illegal trap.
	ILL_PRVOPC	Privileged opcode.
	ILL_PRVREG	Privileged register.
	ILL_COPROC	Coprocessor error.
	ILL_BADSTK	Internal stack error.
SIGFPE	FPE_INTDIV	Integer divide by zero.
	FPE_INTOVF	Integer overflow.
	FPE_FLTDIV	Floating-point divide by zero.
	FPE_FLTOVF	Floating-point overflow.
	FPE_FLTUND	Floating-point underflow.
	FPE_FLTRES	Floating-point inexact result.
	FPE_FLTINV	Invalid floating-point operation.
	FPE_FLTSUB	Subscript out of range.
SIGSEGV	SEGV_MAPERR	Address not mapped to object.
	SEGV_ACCERR	Invalid permissions for mapped object.
SIGBUS	BUS_ADRALN	Invalid address alignment.
	BUS_ADRERR	Non-existent physical address.
	BUS_OBJERR	Object specific hardware error.
SIGTRAP	TRAP_BRKPT	Process breakpoint.
	TRAP_TRACE	Process trace trap.
SIGCHLD	CLD_EXITED	Child has exited.
	CLD_KILLED	Child has terminated abnormally and did not create a core file.
	CLD_DUMPED	Child has terminated abnormally and created a core file.
	CLD_TRAPPED	Traced child has trapped.
	CLD_STOPPED	Child has stopped.
	CLD_CONTINUED	Stopped child has continued.
SIGPOLL	POLL_IN	Data input available.
	POLL_OUT	Output buffers available.
	POLL_MSG	Input message available.
	POLL_ERR	I/O error.
	POLL_PRI	High priority input available.
	POLL_HUP	Device disconnected.
CX Any	SI_USER	Signal sent by *kill*( ).
	SI_QUEUE	Signal sent by the *sigqueue*( ).
	SI_TIMER	Signal generated by expiration of a timer set by *timer_settime*( ).
	SI_ASYNCIO	Signal generated by completion of an asynchronous I/O request.
	SI_MESGQ	Signal generated by arrival of a message on an empty message queue.

XSI Implementations may support additional *si_code* values not included in this list, may generate values included in this list under circumstances other than those described in this list, and may contain extensions or limitations that prevent some values from being generated. Implementations do not generate a different value from the ones described in this list for circumstances described in this list.

In addition, the following signal-specific information shall be available:

Signal	Member	Value
SIGILL SIGFPE	**void * si_addr**	Address of faulting instruction.
SIGSEGV SIGBUS	**void * si_addr**	Address of faulting memory reference.
SIGCHLD	**pid_t si_pid** **int si_status** **uid_t si_uid**	Child process ID. Exit value or signal. Real user ID of the process that sent the signal.
SIGPOLL	**long si_band**	Band event for POLL_IN, POLL_OUT, or POLL_MSG.

For some implementations, the value of *si_addr* may be inaccurate.

The following shall be declared as functions and may also be defined as macros:

```
XSI  void (*bsd_signal(int, void (*)(int)))(int);
CX   int    kill(pid_t, int);
XSI  int    killpg(pid_t, int);
THR  int    pthread_kill(pthread_t, int);
     int    pthread_sigmask(int, const sigset_t *, sigset_t *);
     int    raise(int);
CX   int    sigaction(int, const struct sigaction *restrict,
                struct sigaction *restrict);
     int    sigaddset(sigset_t *, int);
XSI  int    sigaltstack(const stack_t *restrict, stack_t *restrict);
CX   int    sigdelset(sigset_t *, int);
     int    sigemptyset(sigset_t *);
     int    sigfillset(sigset_t *);
XSI  int    sighold(int);
     int    sigignore(int);
     int    siginterrupt(int, int);
CX   int    sigismember(const sigset_t *, int);
     void (*signal(int, void (*)(int)))(int);
XSI  int    sigpause(int);
CX   int    sigpending(sigset_t *);
     int    sigprocmask(int, const sigset_t *restrict, sigset_t *restrict);
RTS  int    sigqueue(pid_t, int, const union sigval);
XSI  int    sigrelse(int);
     void (*sigset(int, void (*)(int)))(int);
CX   int    sigsuspend(const sigset_t *);
RTS  int    sigtimedwait(const sigset_t *restrict, siginfo_t *restrict,
                const struct timespec *restrict);
CX   int    sigwait(const sigset_t *restrict, int *restrict);
RTS  int    sigwaitinfo(const sigset_t *restrict, siginfo_t *restrict);
```

CX Inclusion of the **<signal.h>** header may make visible all symbols from the **<time.h>** header.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<errno.h>, **<stropts.h>**, **<sys/types.h>**, **<time.h>**, **<ucontext.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *alarm*( ), *bsd_signal*( ), *ioctl*( ), *kill*( ), *killpg*( ), *raise*( ), *sigaction*( ), *sigaddset*( ), *sigaltstack*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *siginterrupt*( ), *sigismember*( ), *signal*( ), *sigpending*( ), *sigprocmask*( ), *sigqueue*( ), *sigsuspend*( ), *sigwaitinfo*( ), *wait*( ), *waitid*( )

CHANGE HISTORY

First released in Issue 1.

Issue 5

The DESCRIPTION is updated for alignment with POSIX Realtime Extension and the POSIX Threads Extension.

The default action for SIGURG is changed for i to iii. The function prototype for *sigmask*( ) is removed.

Issue 6

The Open Group Corrigendum U035/2 is applied. In the DESCRIPTION, the wording for abnormal termination is clarified.

The Open Group Corrigendum U028/8 is applied, correcting the prototype for the *sigset*( ) function.

The Open Group Corrigendum U026/3 is applied, correcting the type of the *sigev_notify_function* function member of the **sigevent** structure.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The SIGCHLD, SIGCONT, SIGSTOP, SIGTSTP, SIGTTIN, and SIGTTOU signals are now mandated. This is also a FIPS requirement.
- The **pid_t** definition is mandated.

The RT markings are now changed to RTS to denote that the semantics are part of the Realtime Signals Extension option.

The **restrict** keyword is added to the prototypes for *sigaction*( ), *sigaltstack*( ), *sigprocmask*( ), *sigtimedwait*( ), *sigwait*( ), and *sigwaitinfo*( ).

IEEE PASC Interpretation 1003.1 #85 is applied, adding the statement that symbols from **<time.h>** may be made visible when **<signal.h>** is included. Extensions beyond the ISO C standard are now marked.

NAME

spawn.h — spawn (**ADVANCED REALTIME**)

SYNOPSIS

SPN `#include <spawn.h>`

DESCRIPTION

The **<spawn.h>** header shall define the **posix_spawnattr_t** and **posix_spawn_file_actions_t** types used in performing spawn operations.

The **<spawn.h>** header shall define the flags that may be set in a **posix_spawnattr_t** object using the *posix_spawnattr_setflags*( ) function:

POSIX_SPAWN_RESETIDS
POSIX_SPAWN_SETPGROUP
PS POSIX_SPAWN_SETSCHEDPARAM
POSIX_SPAWN_SETSCHEDULER
POSIX_SPAWN_SETSIGDEF
POSIX_SPAWN_SETSIGMASK

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int   posix_spawn(pid_t *restrict, const char *restrict,
          const posix_spawn_file_actions_t *,
          const posix_spawnattr_t *restrict, char *const [restrict],
          char *const [restrict]);
int   posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *,
          int);
int   posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *,
          int, int);
int   posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict,
          int, const char *restrict, int, mode_t);
int   posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *);
int   posix_spawn_file_actions_init(posix_spawn_file_actions_t *);
int   posix_spawnattr_destroy(posix_spawnattr_t *);
int   posix_spawnattr_getsigdefault(const posix_spawnattr_t *restrict,
          sigset_t *restrict);
int   posix_spawnattr_getflags(const posix_spawnattr_t *restrict,
          short *restrict);
int   posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict,
          pid_t *restrict);
int   posix_spawnattr_getschedparam(const posix_spawnattr_t *restrict,
          struct sched_param *restrict);
int   posix_spawnattr_getschedpolicy(const posix_spawnattr_t *restrict,
          int *restrict);
int   posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict,
          sigset_t *restrict);
int   posix_spawnattr_init(posix_spawnattr_t *);
int   posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict,
          const sigset_t *restrict);
int   posix_spawnattr_setflags(posix_spawnattr_t *, short);
int   posix_spawnattr_setpgroup(posix_spawnattr_t *, pid_t);
```

```
int   posix_spawnattr_setschedparam(posix_spawnattr_t *restrict,
          const struct sched_param *restrict);
int   posix_spawnattr_setschedpolicy(posix_spawnattr_t *, int);
int   posix_spawnattr_setsigmask(posix_spawnattr_t *restrict,
          const sigset_t *restrict);
int   posix_spawnp(pid_t *restrict, const char *restrict,
          const posix_spawn_file_actions_t *,
          const posix_spawnattr_t *restrict,
          char *const [restrict], char *const [restrict]);
```

Inclusion of the **<spawn.h>** header may make visible symbols defined in the **<sched.h>**, **<signal.h>**, and **<sys/types.h>** headers.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sched.h>, **<semaphore.h>**, **<signal.h>**, **<sys/types.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *posix_spawnattr_destroy*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_init*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawn*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_addopen*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawn_file_actions_init*( ), *posix_spawnp*( )

CHANGE HISTORY

First released in Issue 6. Included for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the prototypes for *posix_spawn*( ), *posix_spawn_file_actions_addopen*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setsigmask*( ), and *posix_spawnp*( ).

NAME

stdarg.h — handle variable argument list

SYNOPSIS

```
#include <stdarg.h>

void va_start(va_list ap, argN);
void va_copy(va_list dest, va_list src);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<stdarg.h>** header shall contain a set of macros which allows portable functions that accept variable argument lists to be written. Functions that have variable argument lists (such as *printf*( )) but do not use these macros, are inherently non-portable, as different systems use different argument-passing conventions.

The type **va_list** shall be defined for variables used to traverse the list.

The *va_start*( ) macro is invoked to initialize *ap* to the beginning of the list before any calls to *va_arg*( ).

The *va_copy*( ) macro initializes as a copy of *src*, as if the *va_start*( ) macro had been applied to *dest* followed by the same sequence of uses of the *va_arg*( ) macro as had previously been used to reach the present state of *src*. Neither the *va_copy*( ) nor *va_start*( ) macro shall be invoked to reinitialize *dest* without an intervening invocation of the *va_end*( ) macro for the same *dest*.

The object *ap* may be passed as an argument to another function; if that function invokes the *va_arg*( ) macro with parameter *ap*, the value of *ap* in the calling function is unspecified and shall be passed to the *va_end*( ) macro prior to any further reference to *ap*. The parameter *argN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the ...). If the parameter *argN* is declared with the **register** storage class, with a function type or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

The *va_arg*( ) macro shall return the next argument in the list pointed to by *ap*. Each invocation of *va_arg*( ) modifies *ap* so that the values of successive arguments are returned in turn. The *type* parameter is the type the argument is expected to be. This is the type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by suffixing a `'*'` to type. Different types can be mixed, but it is up to the routine to know what type of argument is expected.

The *va_end*( ) macro is used to clean up; it invalidates *ap* for use (unless *va_start*( ) or *va_copy*( ) is invoked again).

Each invocation of the *va_start*( ) and *va_copy*( ) macros shall be matched by a corresponding invocation of the *va_end*( ) macro in the same function.

Multiple traversals, each bracketed by *va_start*( ) ... *va_end*( ), are possible.

EXAMPLES

This example is a possible implementation of *execl*( ):

```
#include <stdarg.h>
```

```
#define  MAXARGS     31

/*
 * execl is called by
 * execl(file, arg1, arg2, ..., (char *)(0));
 */
int execl(const char *file, const char *args, ...)
{
    va_list ap;
    char *array[MAXARGS];
    int argno = 0;
        va_start(ap, args);
    while (args != 0) {
        array[argno++] = args;
        args = va_arg(ap, const char *);
}
va_end(ap);
return execv(file, array);
}
```

APPLICATION USAGE

It is up to the calling routine to communicate to the called routine how many arguments there are, since it is not always possible for the called routine to determine this in any other way. For example, *execl*( ) is passed a null pointer to signal the end of the list. The *printf*( ) function can tell how many arguments are there by the *format* argument.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *exec*( ), *printf*( )

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

NAME

stdbool.h — boolean type and values

SYNOPSIS

```
#include <stdbool.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<stdbool.h>** header shall define the following macros:

bool	Expands to **_Bool**.
true	Expands to the integer constant 1.
false	Expands to the integer constant 0.
__bool_true_false_are_defined	Expands to the integer constant 1.

An application may undefine and then possibly redefine the macros bool, true, and false.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

The ability to undefine and redefine the macros bool, true, and false is an obsolescent feature and may be withdrawn in the future.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 6. Included for alignment with the ISO/IEC 9899:1999 standard.

NAME

stddef.h — standard type definitions

SYNOPSIS

`#include <stddef.h>`

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<stddef.h>** header shall define the following macros:

NULL	Null pointer constant.
offsetof(*type*, *member-designator*)	Integer constant expression of type **size_t**, the value of which is the offset in bytes to the structure member (*member-designator*), from the beginning of its structure (*type*).

The **<stddef.h>** header shall define the following types:

ptrdiff_t	Signed integer type of the result of subtracting two pointers.
wchar_t	Integer type whose range of values can represent distinct wide-character codes for all members of the largest character set specified among the locales supported by the compilation environment: the null character has the code value 0 and each member of the portable character set has a code value equal to its value when used as the lone character in an integer character constant.
size_t	Unsigned integer type of the result of the *sizeof* operator.

The implementation shall support one or more programming environments in which the widths of **ptrdiff_t**, **size_t**, and **wchar_t** are no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<wchar.h>, **<sys/types.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *confstr*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

NAME

stdint.h — integer types

SYNOPSIS

```
#include <stdint.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<stdint.h>** header shall declare sets of integer types having specified widths, and shall define corresponding sets of macros. It shall also define macros that specify limits of integer types corresponding to types defined in other standard headers.

Note: The ‘‘width’’ of an integer type is the number of bits used to store its value in a pure binary system; the actual type may use more bits than that (for example, a 28-bit type could be stored in 32 bits of actual storage). An N-bit signed type has values in the range -2^{N-1} or $1-2^{N-1}$ to $2^{N-1}-1$, while an N-bit unsigned type has values in the range 0 to $2^{N}-1$.

Types are defined in the following categories:

- Integer types having certain exact widths
- Integer types having at least certain specified widths
- Fastest integer types having at least certain specified widths
- Integer types wide enough to hold pointers to objects
- Integer types having greatest width

(Some of these types may denote the same type.)

Corresponding macros specify limits of the declared types and construct suitable constants.

For each type described herein that the implementation provides, the **<stdint.h>** header shall declare that **typedef** name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, the **<stdint.h>** header shall not declare that **typedef** name, nor shall it define the associated macros. An implementation shall provide those types described as required, but need not provide any of the others (described as optional).

Integer Types

When **typedef** names differing only in the absence or presence of the initial *u* are defined, they shall denote corresponding signed and unsigned types as described in the ISO/IEC 9899:1999 standard, Section 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

In the following descriptions, the symbol N represents an unsigned decimal integer with no leading zeros (for example, 8 or 24, but not 04 or 048).

- Exact-width integer types

  The **typedef** name **int*N*_t** designates a signed integer type with width N, no padding bits, and a two’s-complement representation. Thus, **int8_t** denotes a signed integer type with a width of exactly 8 bits.

  The **typedef** name **uint*N*_t** designates an unsigned integer type with width N. Thus, **uint24_t** denotes an unsigned integer type with a width of exactly 24 bits.

CX The following types are required:

int8_t
int16_t
int32_t
uint8_t
uint16_t
uint32_t

If an implementation provides integer types with width 64 that meet these requirements, then the following types are required:

int64_t
uint64_t

CX In particular, this will be the case if any of the following are true:

— The implementation supports the _POSIX_V6_ILP32_OFFBIG programming environment and the application is being built in the _POSIX_V6_ILP32_OFFBIG programming environment (see the Shell and Utilities volume of IEEE Std 1003.1-200x, *c99*, Programming Environments).

— The implementation supports the _POSIX_V6_LP64_OFF64 programming environment and the application is being built in the _POSIX_V6_LP64_OFF64 programming environment.

— The implementation supports the _POSIX_V6_LPBIG_OFFBIG programming environment and the application is being built in the _POSIX_V6_LPBIG_OFFBIG programming environment.

All other types are of this form optional.

- Minimum-width integer types

The **typedef** name **int_least***N***_t** designates a signed integer type with a width of at least *N*, such that no signed integer type with lesser size has at least the specified width. Thus, **int_least32_t** denotes a signed integer type with a width of at least 32 bits.

The **typedef** name **uint_least***N***_t** designates an unsigned integer type with a width of at least *N*, such that no unsigned integer type with lesser size has at least the specified width. Thus, **uint_least16_t** denotes an unsigned integer type with a width of at least 16 bits.

The following types are required:

int_least8_t
int_least16_t
int_least32_t
int_least64_t
uint_least8_t
uint_least16_t
uint_least32_t
uint_least64_t

All other types of this form are optional.

- Fastest minimum-width integer types

Each of the following types designates an integer type that is usually fastest to operate with among all integer types that have at least the specified width.

The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

The **typedef** name **int_fast***N***_t** designates the fastest signed integer type with a width of at least *N*. The **typedef** name **uint_fast***N***_t** designates the fastest unsigned integer type with a width of at least *N*.

The following types are required:

int_fast8_t
int_fast16_t
int_fast32_t
int_fast64_t
uint_fast8_t
uint_fast16_t
uint_fast32_t
uint_fast64_t

All other types of this form are optional.

- Integer types capable of holding object pointers

The following type designates a signed integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to a pointer to **void**, and the result will compare equal to the original pointer:

intptr_t

The following type designates an unsigned integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to a pointer to **void**, and the result will compare equal to the original pointer:

uintptr_t

XSI On XSI-conformant systems, the **intptr_t** and **uintptr_t** types are required;otherwise, they are optional.

- Greatest-width integer types

The following type designates a signed integer type capable of representing any value of any signed integer type:

intmax_t

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

uintmax_t

These types are required.

Note: Applications can test for optional types by using the corresponding limit macro from **Limits of Specified-Width Integer Types** (on page 316).

Limits of Specified-Width Integer Types

The following macros specify the minimum and maximum limits of the types declared in the **<stdint.h>** header. Each macro name corresponds to a similar type name in **Integer Types** (on page 313).

Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign, except where stated to be exactly the given value.

- Limits of exact-width integer types
  - Minimum values of exact-width signed integer types:

    {INT*N*_MIN} Exactly $-(2^{N-1})$
  - Maximum values of exact-width signed integer types:

    {INT*N*_MAX} Exactly $2^{N-1}-1$
  - Maximum values of exact-width unsigned integer types:

    {UINT*N*_MAX} Exactly $2^{N}-1$
- Limits of minimum-width integer types
  - Minimum values of minimum-width signed integer types:

    {INT_LEAST*N*_MIN} $-(2^{N-1}-1)$
  - Maximum values of minimum-width signed integer types:

    {INT_LEAST*N*_MAX} $2^{N-1}-1$
  - Maximum values of minimum-width unsigned integer types:

    {UINT_LEAST*N*_MAX} $2^{N}-1$
- Limits of fastest minimum-width integer types
  - Minimum values of fastest minimum-width signed integer types:

    {INT_FAST*N*_MIN} $-(2^{N-1}-1)$
  - Maximum values of fastest minimum-width signed integer types:

    {INT_FAST*N*_MAX} $2^{N-1}-1$
  - Maximum values of fastest minimum-width unsigned integer types:

    {UINT_FAST*N*_MAX} $2^{N}-1$
- Limits of integer types capable of holding object pointers
  - Minimum value of pointer-holding signed integer type:

    {INTPTR_MIN} $-(2^{15}-1)$
  - Maximum value of pointer-holding signed integer type:

    {INTPTR_MAX} $2^{15}-1$
  - Maximum value of pointer-holding unsigned integer type:

{UINTPTR_MAX}	$2^{16} - 1$

- Limits of greatest-width integer types
  - Minimum value of greatest-width signed integer type:

    {INTMAX_MIN} $-(2^{63} - 1)$
  - Maximum value of greatest-width signed integer type:

    {INTMAX_MAX} $2^{63} - 1$
  - Maximum value of greatest-width unsigned integer type:

    {UINTMAX_MAX} $2^{64} - 1$

Limits of Other Integer Types

The following macros specify the minimum and maximum limits of integer types corresponding to types defined in other standard headers.

Each instance of these macros shall be replaced by a constant expression suitable for use in **#if** preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Its implementation-defined value shall be equal to or greater in magnitude (absolute value) than the corresponding value given below, with the same sign.

- Limits of **ptrdiff_t**:

{PTRDIFF_MIN}	–65535
{PTRDIFF_MAX}	+65535

- Limits of **sig_atomic_t**:

{SIG_ATOMIC_MIN}	See below.
{SIG_ATOMIC_MAX}	See below.

- Limit of **size_t**:

{SIZE_MAX}	65535

- Limits of **wchar_t**:

{WCHAR_MIN}	See below.
{WCHAR_MAX}	See below.

- Limits of **wint_t**:

{WINT_MIN}	See below.
{WINT_MAX}	See below.

If **sig_atomic_t** (see the **<signal.h>** header) is defined as a signed integer type, the value of {SIG_ATOMIC_MIN} shall be no greater than –127 and the value of {SIG_ATOMIC_MAX} shall be no less than 127; otherwise, **sig_atomic_t** shall be defined as an unsigned integer type, and the value of {SIG_ATOMIC_MIN} shall be 0 and the value of {SIG_ATOMIC_MAX} shall be no less than 255.

If **wchar_t** (see the **<stddef.h>** header) is defined as a signed integer type, the value of {WCHAR_MIN} shall be no greater than –127 and the value of {WCHAR_MAX} shall be no less than 127; otherwise, **wchar_t** shall be defined as an unsigned integer type, and the value of {WCHAR_MIN} shall be 0 and the value of {WCHAR_MAX} shall be no less than 255.

If **wint_t** (see the **<wchar.h>** header) is defined as a signed integer type, the value of {WINT_MIN} shall be no greater than −32767 and the value of {WINT_MAX} shall be no less than 32767; otherwise, **wint_t** shall be defined as an unsigned integer type, and the value of {WINT_MIN} shall be 0 and the value of {WINT_MAX} shall be no less than 65535.

Macros for Integer Constant Expressions

The following macros expand to integer constant expressions suitable for initializing objects that have integer types corresponding to types defined in the **<stdint.h>** header. Each macro name corresponds to a similar type name listed under *Minimum-width integer types* and *Greatest-width integer types*.

Each invocation of one of these macros shall expand to an integer constant expression suitable for use in **#if** preprocessing directives. The type of the expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

The argument in any instance of these macros shall be a decimal, octal, or hexadecimal constant with a value that does not exceed the limits for the corresponding type.

- Macros for minimum-width integer constant expressions

  The macro *INTN_C*(*value*) shall expand to an integer constant expression corresponding to the type **int_least***N***_t**. The macro *UINTN_C*(*value*) shall expand to an integer constant expression corresponding to the type **uint_least***N***_t**. For example, if **uint_least64_t** is a name for the type **unsigned long long**, then *UINT64_C*(0x123) might expand to the integer constant 0x123ULL.

- Macros for greatest-width integer constant expressions

  The following macro expands to an integer constant expression having the value specified by its argument and the type **intmax_t**:

  INTMAX_C(*value*)

  The following macro expands to an integer constant expression having the value specified by its argument and the type **uintmax_t**:

  UINTMAX_C(*value*)

APPLICATION USAGE

None.

RATIONALE

The **<stdint.h>** header is a subset of the **<inttypes.h>** header more suitable for use in freestanding environments, which might not support the formatted I/O functions. In some environments, if the formatted conversion support is not wanted, using this header instead of the **<inttypes.h>** header avoids defining such a large number of macros.

As a consequence of adding **int8_t** the following are true:

- A byte is exactly 8 bits.
- {CHAR_BIT} has the value 8, {SCHAR_MAX} has the value 127, {SCHAR_MIN} has the value −127 or −128, and {UCHAR_MAX} has the value 255.

FUTURE DIRECTIONS

typedef names beginning with **int** or **uint** and ending with _t may be added to the types defined in the **<stdint.h>** header. Macro names beginning with INT or UINT and ending with _MAX, _MIN, or _C may be added to the macros defined in the **<stdint.h>** header.

SEE ALSO

<signal.h>, **<stddef.h>**, **<wchar.h>**, **<inttypes.h>**

CHANGE HISTORY

First released in Issue 6. Included for alignment with the ISO / IEC 9899: 1999 standard. |

ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

stdio.h — standard buffered input/output

SYNOPSIS

```
#include <stdio.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<stdio.h>** header shall define the following macros as positive integer constant expressions:

BUFSIZ	Size of **<stdio.h>** buffers.
_IOFBF	Input/output fully buffered.
_IOLBF	Input/output line buffered.
_IONBF	Input/output unbuffered.
CX L_ctermid	Maximum size of character array to hold *ctermid*( ) output.
L_tmpnam	Maximum size of character array to hold *tmpnam*( ) output.
SEEK_CUR	Seek relative to current position.
SEEK_END	Seek relative to end-of-file.
SEEK_SET	Seek relative to start-of-file.

The following macros shall be defined as positive integer constant expressions which denote implementation limits:

{FILENAME_MAX}	Maximum size in bytes of the longest filename string that the implementation guarantees can be opened.
{FOPEN_MAX}	Number of streams which the implementation guarantees can be open simultaneously. The value is at least eight.
{TMP_MAX}	Minimum number of unique filenames generated by *tmpnam*( ). Maximum number of times an application can call *tmpnam*( ) reliably. The value of {TMP_MAX} is at least 25. XSI On XSI-conformant systems, the value of {TMP_MAX} is at least 10,000.

The following macro name shall be defined as a negative integer constant expression:

EOF	End-of-file return value.

The following macro name shall be defined as a null pointer constant:

NULL	Null pointer.

The following macro name shall be defined as a string constant:

XSI P_tmpdir	Default directory prefix for *tempnam*( ).

The following shall be defined as expressions of type ‘‘pointer to **FILE**’’ that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams:

stderr	Standard error output stream.
stdin	Standard input stream.

stdout	Standard output stream.

The following data types shall be defined through **typedef**:

FILE	A structure containing information about a file.
fpos_t	A non-array type containing all information needed to specify uniquely every position within a file.
va_list	As described in **<stdarg.h>**.
size_t	As described in **<stddef.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
void     clearerr(FILE *);
char    *ctermid(char *);
int      fclose(FILE *);
FILE    *fdopen(int, const char *);
int      feof(FILE *);
int      ferror(FILE *);
int      fflush(FILE *);
int      fgetc(FILE *);
int      fgetpos(FILE *restrict, fpos_t *restrict);
char    *fgets(char *restrict, int, FILE *restrict);
int      fileno(FILE *);
void     flockfile(FILE *);
FILE    *fopen(const char *restrict, const char *restrict);
int      fprintf(FILE *restrict, const char *restrict, ...);
int      fputc(int, FILE *);
int      fputs(const char *restrict, FILE *restrict);
size_t   fread(void *restrict, size_t, size_t, FILE *restrict);
FILE    *freopen(const char *restrict, const char *restrict,
             FILE *restrict);
int      fscanf(FILE *restrict, const char *restrict, ...);
int      fseek(FILE *, long, int);
int      fseeko(FILE *, off_t, int);
int      fsetpos(FILE *, const fpos_t *);
long     ftell(FILE *);
off_t    ftello(FILE *);
int      ftrylockfile(FILE *);
void     funlockfile(FILE *);
size_t   fwrite(const void *restrict, size_t, size_t, FILE *restrict);
int      getc(FILE *);
int      getchar(void);
int      getc_unlocked(FILE *);
int      getchar_unlocked(void);
char    *gets(char *);
int      pclose(FILE *);
void     perror(const char *);
FILE    *popen(const char *, const char *);
int      printf(const char *restrict, ...);
int      putc(int, FILE *);
int      putchar(int);
```

```
int      putc_unlocked(int, FILE *);
int      putchar_unlocked(int);
int      puts(const char *);
int      remove(const char *);
int      rename(const char *, const char *);
void     rewind(FILE *);
int      scanf(const char *restrict, ...);
void     setbuf(FILE *restrict, char *restrict);
int      setvbuf(FILE *restrict, char *restrict, int, size_t);
int      snprintf(char *restrict, size_t, const char *restrict, ...);
int      sprintf(char *restrict, const char *restrict, ...);
int      sscanf(const char *restrict, const char *restrict, int ...);
char    *tempnam(const char *, const char *);
FILE    *tmpfile(void);
char    *tmpnam(char *);
int      ungetc(int, FILE *);
int      vfprintf(FILE *restrict, const char *restrict, va_list);
int      vfscanf(FILE *restrict, const char *restrict, va_list);
int      vprintf(const char *restrict, va_list);
int      vscanf(const char *restrict, va_list);
int      vsnprintf(char *restrict, size_t, const char *restrict, va_list;
int      vsprintf(char *restrict, const char *restrict, va_list);
int      vsscanf(const char *restrict, const char *restrict, va_list arg);
```

XSI Inclusion of the **<stdio.h>** header may also make visible all symbols from **<stddef.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *clearerr*( ), *ctermid*( ), *fclose*( ), *fdopen*( ), *fgetc*( ), *fgetpos*( ), *ferror*( ), *feof*( ), *fflush*( ), *fgets*( ), *fileno*( ), *flockfile*( ), *fopen*( ), *fputc*( ), *fputs*( ), *fread*( ), *freopen*( ), *fseek*( ), *fsetpos*( ), *ftell*( ), *fwrite*( ), *getc*( ), *getc_unlocked*( ), *getwchar*( ), *getchar*( ), *getopt*( ), *gets*( ), *pclose*( ), *perror*( ), *popen*( ), *printf*( ), *putc*( ), *putchar*( ), *puts*( ), *putwchar*( ), *remove*( ), *rename*( ), *rewind*( ), *scanf*( ), *setbuf*( ), *setvbuf*( ), *sscanf*( ), *stdin*, *system*( ), *tempnam*( ), *tmpfile*( ), *tmpnam*( ), *ungetc*( ), *vfscanf*( ), *vscanf*( ), *vprintf*( ), *vsscanf*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Large File System extensions are added.

The constant L_cuserid and the external variables *optarg*, *opterr*, *optind*, and *optopt* are marked as extensions and LEGACY.

The *cuserid*( ) and *getopt*( ) functions are marked LEGACY.

Issue 6

The constant L_cuserid and the external variables *optarg*, *opterr*, *optind*, and *optopt* are removed as they were previously marked LEGACY.

The *cuserid*( ), *getopt*( ), and *getw*( ) functions are removed as they were previously marked LEGACY.

Several functions are marked as part of the _POSIX_THREAD_SAFE_FUNCTIONS option.

This reference page is updated to align with the ISO/IEC 9899:1999 standard. Note that the description of the **fpos_t** type is now explicitly updated to exclude array types.

Extensions beyond the ISO C standard are now marked.

NAME

stdlib.h — standard library definitions

SYNOPSIS

```
#include <stdlib.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<stdlib.h>** header shall define the following macros:

EXIT_FAILURE	Unsuccessful termination for *exit*( ); evaluates to a non-zero value.
EXIT_SUCCESS	Successful termination for *exit*( ); evaluates to 0.
NULL	Null pointer.
{RAND_MAX}	Maximum value returned by *rand*( ); at least 32,767.
{MB_CUR_MAX}	Integer expression whose value is the maximum number of bytes in a character specified by the current locale.

The following data types shall be defined through **typedef**:

div_t	Structure type returned by the *div*( ) function.
ldiv_t	Structure type returned by the *ldiv*( ) function.
lldiv_t	Structure type returned by the *lldiv*( ) function.
size_t	As described in **<stddef.h>**.
wchar_t	As described in **<stddef.h>**.

In addition, the following symbolic names and macros shall be defined as in **<sys/wait.h>**, for use in decoding the return value from *system*( ):

XSI

WNOHANG
WUNTRACED
WEXITSTATUS
WIFEXITED
WIFSIGNALED
WIFSTOPPED
WSTOPSIG
WTERMSIG

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
void          _Exit(int);
long          a64l(const char *);
void          abort(void);
int           abs(int);
int           atexit(void (*)(void));
double        atof(const char *);
int           atoi(const char *);
long          atol(const char *);
```

```
     long long     atoll(const char *);
     void         *bsearch(const void *, const void *, size_t, size_t,
                       int (*)(const void *, const void *));
     void         *calloc(size_t, size_t);
     div_t         div(int, int);
XSI  double        drand48(void);
     char         *ecvt(double, int, int *restrict, int *restrict); (LEGACY)
     double        erand48(unsigned short[3]);
     void          exit(int);
XSI  char         *fcvt(double, int, int *restrict, int *restrict); (LEGACY)
     void          free(void *);
XSI  char         *gcvt(double, int, char *); (LEGACY)
     char         *getenv(const char *);
XSI  int           getsubopt(char **, char *const *, char **);
     int           grantpt(int);
     char         *initstate(unsigned, char *, size_t);
     long          jrand48(unsigned short[3]);
     char         *l64a(long);
     long          labs(long);
XSI  void          lcong48(unsigned short[7]);
     ldiv_t        ldiv(long, long);
     long long     llabs(long long);
     lldiv_t       lldiv(long long, long long);
XSI  long          lrand48(void);
     void         *malloc(size_t);
     int           mblen(const char *, size_t);
     size_t        mbstowcs(wchar_t *restrict, const char *restrict, size_t);
     int           mbtowc(wchar_t *restrict, const char *restrict, size_t);
XSI  char         *mktemp(char *); (LEGACY)
     int           mkstemp(char *);
     long          mrand48(void);
     long          nrand48(unsigned short[3]);
ADV  int           posix_memalign(void **, size_t, size_t);
XSI  int           posix_openpt(int);
     char         *ptsname(int);
     int           putenv(char *);
     void          qsort(void *, size_t, size_t, int (*)(const void *,
                       const void *));
     int           rand(void);
TSF  int           rand_r(unsigned *);
XSI  long          random(void);
     void         *realloc(void *, size_t);
XSI  char         *realpath(const char *restrict, char *restrict);
     unsigned short seed48(unsigned short[3]);
CX   int           setenv(const char *, const char *, int);
XSI  void          setkey(const char *);
     char         *setstate(const char *);
     void          srand(unsigned);
XSI  void          srand48(long);
     void          srandom(unsigned);
     double        strtod(const char *restrict, char **restrict);
     float         strtof(const char *restrict, char **restrict);
```

```
    long          strtol(const char *restrict, char **restrict, int);
    long double   strtold(const char *restrict, char **restrict);
    long long     strtoll(const char *restrict, char **restrict, int);
    unsigned long strtoul(const char *restrict, char **restrict, int);
    long long     strtoull(const char *restrict, char **restrict, int);
    int           system(const char *);
XSI int           unlockpt(int);
CX  int           unsetenv(const char *);
    size_t        wcstombs(char *restrict, const wchar_t *restrict, size_t);
    int           wctomb(char *, wchar_t);
```

XSI Inclusion of the **<stdlib.h>** header may also make visible all symbols from **<stddef.h>**, **<limits.h>**, **<math.h>**, and **<sys/wait.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *_Exit*( ), *a64l*( ), *abort*( ), *abs*( ), *atexit*( ), *atof*( ), *atoi*( ), *atol*( ), *atoll*( ), *bsearch*( ), *calloc*( ), *div*( ), *drand48*( ), *erand48*( ), *exit*( ), *free*( ), *getenv*( ), *getsubopt*( ), *grantpt*( ), *initstate*( ), *jrand48*( ), *l64a*( ), *labs*( ), *lcong48*( ), *ldiv*( ), *llabs*( ), *lldiv*( ), *lrand48*( ), *malloc*( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *mkstemp*( ), *mrand48*( ), *nrand48*( ), *posix_memalign*( ), *ptsname*( ), *putenv*( ), *qsort*( ), *rand*( ), *realloc*( ), *realpath*( ), *setstate*( ), *srand*( ), *srand48*( ), *srandom*( ), *strtod*( ), *strtof*( ), *strtol*( ), *strtold*( ), *strtoll*( ), *strtoul*( ), *strtoull*( ), *unlockpt*( ), *wcstombs*( ), *wctomb*( )

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

The *ttyslot*( ) and *valloc*( ) functions are marked LEGACY.

The type of the third argument to *initstate*( ) is changed from **int** to **size_t**. The type of the return value from *setstate*( ) is changed from **char** to **char ***, and the type of the first argument is changed from **char *** to **const char ***.

Issue 6

The Open Group Corrigendum U021/1 is applied, correcting the prototype for *realpath*( ) to be consistent with the reference page.

The Open Group Corrigendum U028/13 is applied, correcting the prototype for *putenv*( ) to be consistent with the reference page.

The *rand_r*( ) function is marked as part of the _POSIX_THREAD_SAFE_FUNCTIONS option.

Function prototypes for *setenv*( ) and *unsetenv*( ) are added.

The *posix_memalign*( ) function is added for alignment with IEEE Std 1003.1d-1999.

This reference page is updated to align with the ISO/IEC 9899:1999 standard.

The *ecvt*( ), *fcvt*( ), *gcvt*( ), and *mktemp*( ) functions are marked LEGACY.

The *ttyslot*( ) and *valloc*( ) functions are removed as they were previously marked LEGACY.

Extensions beyond the ISO C standard are now marked.

NAME

string.h — string operations

SYNOPSIS

```
#include <string.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<string.h>** header shall define the following:

NULL	Null pointer constant.
size_t	As described in **<stddef.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
XSI void    *memccpy(void *restrict, const void *restrict, int, size_t);
    void    *memchr(const void *, int, size_t);
    int      memcmp(const void *, const void *, size_t);
    void    *memcpy(void *restrict, const void *restrict, size_t);
    void    *memmove(void *, const void *, size_t);
    void    *memset(void *, int, size_t);
    char    *strcat(char *restrict, const char *restrict);
    char    *strchr(const char *, int);
    int      strcmp(const char *, const char *);
    int      strcoll(const char *, const char *);
    char    *strcpy(char *restrict, const char *restrict);
    size_t   strcspn(const char *, const char *);
XSI char    *strdup(const char *);
    char    *strerror(int);
    size_t   strlen(const char *);
    char    *strncat(char *restrict, const char *restrict, size_t);
    int      strncmp(const char *, const char *, size_t);
    char    *strncpy(char *restrict, const char *restrict, size_t);
    char    *strpbrk(const char *, const char *);
    char    *strrchr(const char *, int);
    size_t   strspn(const char *, const char *);
    char    *strstr(const char *, const char *);
    char    *strtok(char *restrict, const char *restrict);
TSF char    *strtok_r(char *, const char *, char **);
    size_t   strxfrm(char *restrict, const char *restrict, size_t);
```

XSI Inclusion of the **<string.h>** header may also make visible all symbols from **<stddef.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *memccpy*( ), *memchr*( ), *memcmp*( ), *memcpy*( ), *memmove*( ), *memset*( ), *strcat*( ), *strchr*( ), *strcmp*( ), *strcoll*( ), *strcpy*( ), *strcspn*( ), *strdup*( ), *strerror*( ), *strlen*( ), *strncat*( ), *strncmp*( ), *strncpy*( ), *strpbrk*( ), *strrchr*( ), *strspn*( ), *strstr*( ), *strtok*( ), *strxfrm*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The *strtok_r*( ) function is marked as part of the _POSIX_THREAD_SAFE_FUNCTIONS option.

This reference page is updated to align with the ISO / IEC 9899: 1999 standard.

NAME

strings.h — string operations

SYNOPSIS

XSI

```
#include <strings.h>
```

DESCRIPTION

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    bcmp(const void *, const void *, size_t); (LEGACY)
void   bcopy(const void *, void *, size_t); (LEGACY)
void   bzero(void *, size_t); (LEGACY)
int    ffs(int);
char  *index(const char *, int); (LEGACY)
char  *rindex(const char *, int); (LEGACY)
int    strcasecmp(const char *, const char *);
int    strncasecmp(const char *, const char *, size_t);
```

The **size_t** type shall be defined through **typedef** as described in **<stddef.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<stddef.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *ffs*( ), *strcasecmp*( ), *strncasecmp*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

The Open Group Corrigendum U021/2 is applied, correcting the prototype for *index*( ) to be consistent with the reference page.

The *bcmp*( ), *bcopy*( ), *bzero*( ), *index*( ), and *rindex*( ) functions are marked LEGACY.

NAME

stropts.h — STREAMS interface (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>
```

DESCRIPTION

The **<stropts.h>** header shall define the **bandinfo** structure that includes at least the following members:

`unsigned char`	`bi_pri`	Priority band.
`int`	`bi_flag`	Flushing type.

The **<stropts.h>** header shall define the **strpeek** structure that includes at least the following members:

`struct strbuf`	`ctlbuf`	The control portion of the message.
`struct strbuf`	`databuf`	The data portion of the message.
`t_uscalar_t`	`flags`	RS_HIPRI or 0.

The **<stropts.h>** header shall define the **strbuf** structure that includes at least the following members:

`int`	`maxlen`	Maximum buffer length.
`int`	`len`	Length of data.
`char`	`*buf`	Pointer to buffer.

The **<stropts.h>** header shall define the **strfdinsert** structure that includes at least the following members:

`struct strbuf`	`ctlbuf`	The control portion of the message.
`struct strbuf`	`databuf`	The data portion of the message.
`t_uscalar_t`	`flags`	RS_HIPRI or 0.
`int`	`fildes`	File descriptor of the other STREAM.
`int`	`offset`	Relative location of the stored value.

The **<stropts.h>** header shall define the **strioctl** structure that includes at least the following members:

`int`	`ic_cmd`	*ioctl*( ) command.
`int`	`ic_timout`	Timeout for response.
`int`	`ic_len`	Length of data.
`char`	`*ic_dp`	Pointer to buffer.

The **<stropts.h>** header shall define the **strrecvfd** structure that includes at least the following members:

`int`	`fda`	Received file descriptor.
`uid_t`	`uid`	UID of sender.
`gid_t`	`gid`	GID of sender.

The **uid_t** and **gid_t** types shall be defined through **typedef** as described in **<sys/types.h>**.

The **<stropts.h>** header shall define the **t_scalar_t** and **t_uscalar_t** types respectively as signed and unsigned opaque types of equal length of at least 32 bits.

The **<stropts.h>** header shall define the **str_list** structure that includes at least the following members:

`int`	`sl_nmods`	Number of STREAMS module names.
`struct str_mlist`	`*sl_modlist`	STREAMS module names.

The **<stropts.h>** header shall define the **str_mlist** structure that includes at least the following member:

`char`	`l_name[FMNAMESZ+1]`	A STREAMS module name.

At least the following macros shall be defined for use as the *request* argument to *ioctl*( ):

I_PUSH	Push a STREAMS module.
I_POP	Pop a STREAMS module.
I_LOOK	Get the top module name.
I_FLUSH	Flush a STREAM.
I_FLUSHBAND	Flush one band of a STREAM.
I_SETSIG	Ask for notification signals.
I_GETSIG	Retrieve current notification signals.
I_FIND	Look for a STREAMS module.
I_PEEK	Peek at the top message on a STREAM.
I_SRDOPT	Set the read mode.
I_GRDOPT	Get the read mode.
I_NREAD	Size the top message.
I_FDINSERT	Send implementation-defined information about another STREAM.
I_STR	Send a STREAMS *ioctl*( ).
I_SWROPT	Set the write mode.
I_GWROPT	Get the write mode.
I_SENDFD	Pass a file descriptor through a STREAMS pipe.
I_RECVFD	Get a file descriptor sent via I_SENDFD.
I_LIST	Get all the module names on a STREAM.
I_ATMARK	Is the top message ‘‘marked’’?
I_CKBAND	See if any messages exist in a band.
I_GETBAND	Get the band of the top message on a STREAM.
I_CANPUT	Is a band writable?
I_SETCLTIME	Set close time delay.
I_GETCLTIME	Get close time delay.
I_LINK	Connect two STREAMs.
I_UNLINK	Disconnect two STREAMs.
I_PLINK	Persistently connect two STREAMs.
I_PUNLINK	Dismantle a persistent STREAMS link.

At least the following macros shall be defined for use with I_LOOK:

FMNAMESZ	The minimum size in bytes of the buffer referred to by the *arg* argument.

At least the following macros shall be defined for use with I_FLUSH:

FLUSHR	Flush read queues.
FLUSHW	Flush write queues.
FLUSHRW	Flush read and write queues.

At least the following macros shall be defined for use with I_SETSIG:

S_RDNORM	A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue.
S_RDBAND	A message with a non-zero priority band has arrived at the head of a STREAM head read queue.
S_INPUT	A message, other than a high-priority message, has arrived at the head of a STREAM head read queue.
S_HIPRI	A high-priority message is present on a STREAM head read queue.
S_OUTPUT	The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
S_WRNORM	Equivalent to S_OUTPUT.
S_WRBAND	The write queue for a non-zero priority band just below the STREAM head is no longer full.
S_MSG	A STREAMS signal message that contains the SIGPOLL signal reaches the front of the STREAM head read queue.
S_ERROR	Notification of an error condition reaches the STREAM head.
S_HANGUP	Notification of a hangup reaches the STREAM head.
S_BANDURG	When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.

At least the following macros shall be defined for use with I_PEEK:

RS_HIPRI	Only look for high-priority messages.

At least the following macros shall be defined for use with I_SRDOPT:

RNORM	Byte-STREAM mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.
RPROTNORM	Fail *read*( ) with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.
RPROTDAT	Deliver the control part of a message as data when a process issues a *read*( ).
RPROTDIS	Discard the control part of a message, delivering any data part, when a process issues a *read*( ).

At least the following macros shall be defined for use with I_SWOPT:

SNDZERO	Send a zero-length message downstream when a *write*( ) of 0 bytes occurs.

At least the following macros shall be defined for use with I_ATMARK:

ANYMARK	Check if the message is marked.
LASTMARK	Check if the message is the last one marked on the queue.

At least the following macros shall be defined for use with I_UNLINK:

MUXID_ALL	Unlink all STREAMs linked to the STREAM associated with *fildes*.

The following macros shall be defined for *getmsg*( ), *getpmsg*( ), *putmsg*( ), and *putpmsg*( ):

MSG_ANY	Receive any message.
MSG_BAND	Receive message from specified band.
MSG_HIPRI	Send/receive high-priority message.
MORECTL	More control information is left in message.
MOREDATA	More data is left in message.

The **<stropts.h>** header may make visible all of the symbols from **<unistd.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    isastream(int);
int    getmsg(int, struct strbuf *restrict, struct strbuf *restrict,
           int *restrict);
int    getpmsg(int, struct strbuf *restrict, struct strbuf *restrict,
           int *restrict, int *restrict);
int    ioctl(int, int, ... );
int    putmsg(int, const struct strbuf *, const struct strbuf *, int);
int    putpmsg(int, const struct strbuf *, const struct strbuf *, int,
           int);
int    fattach(int, const char *);
int    fdetach(const char *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *close*( ), *fcntl*( ), *getmsg*( ), *ioctl*( ), *open*( ), *pipe*( ), *read*( ), *poll*( ), *putmsg*( ), *signal*( ), *write*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

The *flags* member of the **strpeek** and **strfdinsert** structures are changed from **type long** to **t_uscalar_t**.

Issue 6

This header is marked as part of the XSI STREAMS Option Group.

The **restrict** keyword is added to the prototypes for *getmsg*( ) and *getpmsg*( ).

NAME

sys/ipc.h — XSI interprocess communication access structure

SYNOPSIS

XSI

```
#include <sys/ipc.h>
```

DESCRIPTION

The **<sys/ipc.h>** header is used by three mechanisms for XSI interprocess communication (IPC): messages, semaphores, and shared memory. All use a common structure type, **ipc_perm** to pass information used in determining permission to perform an IPC operation.

The **ipc_perm** structure shall contain the following members:

`uid_t`	`uid`	Owner's user ID.
`gid_t`	`gid`	Owner's group ID.
`uid_t`	`cuid`	Creator's user ID.
`gid_t`	`cgid`	Creator's group ID.
`mode_t`	`mode`	Read/write permission.

The **uid_t**, **gid_t**, **mode_t**, and **key_t** types shall be defined as described in **<sys/types.h>**.

Definitions shall be provided for the following constants:

Mode bits:

IPC_CREAT	Create entry if key does not exist.
IPC_EXCL	Fail if key exists.
IPC_NOWAIT	Error if request must wait.

Keys:

IPC_PRIVATE	Private key.

Control commands:

IPC_RMID	Remove identifier.
IPC_SET	Set options.
IPC_STAT	Get options.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided.

```
key_t  ftok(const char *, int);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *ftok*( )

CHANGE HISTORY

First released in Issue 2. Derived from System V Release 2.0.

NAME

sys/mman.h — memory management declarations

SYNOPSIS

`#include <sys/mman.h>`

DESCRIPTION

The **<sys/mman.h>** header shall be supported if the implementation supports at least one of the following options:

- MF The Memory Mapped Files option
- SHM The Shared Memory Objects option
- ML The Process Memory Locking option
- MPR The Memory Protection option
- TYM The Typed Memory Objects option
- SIO The Synchronized Input and Output option
- ADV The Advisory Information option
- TYM The Typed Memory Objects option

MC2 If one or more of the Advisory Information, Memory Mapped Files, or Shared Memory Objects options are supported, the following protection options shall be defined:

MC2	PROT_READ	Page can be read.
MC2	PROT_WRITE	Page can be written.
MC2	PROT_EXEC	Page can be executed.
MC2	PROT_NONE	Page cannot be accessed.

The following *flag* options shall be defined:

MF|SHM	MAP_SHARED	Share changes.
MF|SHM	MAP_PRIVATE	Changes are private.
MF|SHM	MAP_FIXED	Interpret *addr* exactly.

The following flags shall be defined for *msync*( ):

MF|SIO	MS_ASYNC	Perform asynchronous writes.
MF|SIO	MS_SYNC	Perform synchronous writes.
MF|SIO	MS_INVALIDATE	Invalidate mappings.

ML The following symbolic constants shall be defined for the *mlockall*( ) function:

ML	MCL_CURRENT	Lock currently mapped pages.
ML	MCL_FUTURE	Lock pages that become mapped.

MF|SHM The symbolic constant MAP_FAILED shall be defined to indicate a failure from the *mmap*( ) function.

MC1 If the Advisory Information and either the Memory Mapped Files or Shared Memory Objects options are supported, values for *advice* used by *posix_madvise*( ) shall be defined as follows:

POSIX_MADV_NORMAL
The application has no advice to give on its behavior with respect to the specified range. It

is the default characteristic if no advice is given for a range of memory.

POSIX_MADV_SEQUENTIAL
The application expects to access the specified range sequentially from lower addresses to higher addresses.

POSIX_MADV_RANDOM
The application expects to access the specified range in a random order.

POSIX_MADV_WILLNEED
The application expects to access the specified range in the near future.

POSIX_MADV_DONTNEED
The application expects that it will not access the specified range in the near future.

TYM The following flags shall be defined for *posix_typed_mem_open*( ):

POSIX_TYPED_MEM_ALLOCATE
Allocate on *mmap*( ).

POSIX_TYPED_MEM_ALLOCATE_CONTIG
Allocate contiguously on *mmap*( ).

POSIX_TYPED_MEM_MAP_ALLOCATABLE Map on *mmap*( ), without affecting allocatability.

The **mode_t**, **off_t**, and **size_t** types shall be defined as described in **<sys/types.h>**.

TYM The **<sys/mman.h>** header shall define the structure **posix_typed_mem_info**, which includes at least the following member:

`size_t`	`posix_tmi_length`	Maximum length which may be allocated from a typed memory object.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
ML       int    mlock(const void *, size_t);
         int    mlockall(int);
MF|SHM   void  *mmap(void *, size_t, int, int, int, off_t);
MPR      int    mprotect(void *, size_t, int);
MF|SIO   int    msync(void *, size_t, int);
ML       int    munlock(const void *, size_t);
         int    munlockall(void);
MF|SHM   int    munmap(void *, size_t);
ADV      int    posix_madvise(void *, size_t, int);
TYM      int    posix_mem_offset(const void *restrict, size_t, off_t *restrict,
                    size_t *restrict, int *restrict);
         int    posix_typed_mem_get_info(int, struct posix_typed_mem_info *);
         int    posix_typed_mem_open(const char *, int, int);
SHM      int    shm_open(const char *, int, mode_t);
         int    shm_unlink(const char *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *mlock*( ), *mlockall*( ), *mmap*( ), *mprotect*( ), *msync*( ), *munlock*( ), *munlockall*( ), *munmap*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *posix_typed_mem_open*( ), *shm_open*( ), *shm_unlink*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Updated for alignment with the POSIX Realtime Extension.

Issue 6

The **<sys/mman.h>** header is marked as dependent on support for either the _POSIX_MAPPED_FILES, _POSIX_MEMLOCK, or _POSIX_SHARED_MEMORY options.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The TYM margin code is added to the list of margin codes for the **<sys/mman.h>** header line, as well as for other lines.
- The POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEM_ALLOCATE_CONTIG, and POSIX_TYPED_MEM_MAP_ALLOCATABLE flags are added.
- The **posix_tmi_length** structure is added.
- The *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), and *posix_typed_mem_open*( ) functions are added.

The **restrict** keyword is added to the prototype for *posix_mem_offset*( ).

IEEE PASC Interpretation 1003.1 #102 is applied adding the prototype for *posix_madvise*( ).

NAME

sys/msg.h — XSI message queue structures

SYNOPSIS

XSI

```
#include <sys/msg.h>
```

DESCRIPTION

The **<sys/msg.h>** header shall define the following constant and members of the structure **msqid_ds**.

The following data types shall be defined through **typedef**:

msgqnum_t	Used for the number of messages in the message queue.
msglen_t	Used for the number of bytes allowed in a message queue.

These types shall be unsigned integer types that are able to store values at least as large as a type **unsigned short**.

Message operation flag:

MSG_NOERROR	No error if big message.

The **msqid_ds** structure shall contain the following members:

`struct ipc_perm`	`msg_perm`	Operation permission structure.
`msgqnum_t`	`msg_qnum`	Number of messages currently on queue.
`msglen_t`	`msg_qbytes`	Maximum number of bytes allowed on queue.
`pid_t`	`msg_lspid`	Process ID of last *msgsnd*( ).
`pid_t`	`msg_lrpid`	Process ID of last *msgrcv*( ).
`time_t`	`msg_stime`	Time of last *msgsnd*( ).
`time_t`	`msg_rtime`	Time of last *msgrcv*( ).
`time_t`	`msg_ctime`	Time of last change.

The **pid_t**, **time_t**, **key_t**, **size_t**, and **ssize_t** types shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int      msgctl(int, int, struct msqid_ds *);
int      msgget(key_t, int);
ssize_t  msgrcv(int, void *, size_t, long, int);
int      msgsnd(int, const void *, size_t, int);
```

In addition, all of the symbols from **<sys/ipc.h>** shall be defined when this header is included.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( )

CHANGE HISTORY

First released in Issue 2. Derived from System V Release 2.0.

NAME

sys/resource.h — definitions for XSI resource operations

SYNOPSIS

XSI
```
#include <sys/resource.h>
```

DESCRIPTION

The **<sys/resource.h>** header shall define the following symbolic constants as possible values of the *which* argument of *getpriority*( ) and *setpriority*( ):

PRIO_PROCESS	Identifies the *who* argument as a process ID.
PRIO_PGRP	Identifies the *who* argument as a process group ID.
PRIO_USER	Identifies the *who* argument as a user ID.

The following type shall be defined through **typedef**:

rlim_t	Unsigned integer type used for limit values.

The following symbolic constants shall be defined:

RLIM_INFINITY	A value of **rlim_t** indicating no limit.
RLIM_SAVED_MAX	A value of type **rlim_t** indicating an unrepresentable saved hard limit.
RLIM_SAVED_CUR	A value of type **rlim_t** indicating an unrepresentable saved soft limit.

On implementations where all resource limits are representable in an object of type **rlim_t**, RLIM_SAVED_MAX and RLIM_SAVED_CUR need not be distinct from RLIM_INFINITY.

The following symbolic constants shall be defined as possible values of the *who* parameter of *getrusage*( ):

RUSAGE_SELF	Returns information about the current process.
RUSAGE_CHILDREN	Returns information about children of the current process.

The **<sys/resource.h>** header shall define the **rlimit** structure that includes at least the following members:

`rlim_t rlim_cur`	The current (soft) limit.
`rlim_t rlim_max`	The hard limit.

The **<sys/resource.h>** header shall define the **rusage** structure that includes at least the following members:

`struct timeval ru_utime`	User time used.
`struct timeval ru_stime`	System time used.

The **timeval** structure shall be defined as described in **<sys/time.h>**.

The following symbolic constants shall be defined as possible values for the *resource* argument of *getrlimit*( ) and *setrlimit*( ):

RLIMIT_CORE	Limit on size of core dump file.
RLIMIT_CPU	Limit on CPU time per process.
RLIMIT_DATA	Limit on data segment size.
RLIMIT_FSIZE	Limit on file size.

RLIMIT_NOFILE	Limit on number of open files.
RLIMIT_STACK	Limit on stack size.
RLIMIT_AS	Limit on address space size.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
int  getpriority(int, id_t);
int  getrlimit(int, struct rlimit *);
int  getrusage(int, struct rusage *);
int  setpriority(int, id_t, int);
int  setrlimit(int, const struct rlimit *);
```

The **id_t** type shall be defined through **typedef** as described in **<sys/types.h>**.

Inclusion of the **<sys/resource.h>** header may also make visible all symbols from **<sys/time.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/time.h>, **<sys/types.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *getpriority*( ), *getrusage*( ), *getrlimit*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Large File System extensions are added.

NAME

sys/select.h — select types

SYNOPSIS

```
#include <sys/select.h>
```

DESCRIPTION

The **<sys/select.h>** header shall define the **timeval** structure that includes at least the following members:

`time_t`	`tv_sec`	Seconds.
`suseconds_t`	`tv_usec`	Microseconds.

The **time_t** and **suseconds_t** types shall be defined as described in **<sys/types.h>**.

The **sigset_t** type shall be defined as described in **<signal.h>**.

The **timespec** structure shall be defined as described in **<time.h>**.

The **<sys/select.h>** header shall define the **fd_set** type as a structure.

Each of the following may be declared as a function, or defined as a macro, or both:

void *FD_CLR*(int *fd*, fd_set **fdset*)
Clears the bit for the file descriptor *fd* in the file descriptor set *fdset*.

int *FD_ISSET*(int *fd*, fd_set **fdset*)
Returns a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set by *fdset*, and 0 otherwise.

void *FD_SET*(int *fd*, fd_set **fdset*)
Sets the bit for the file descriptor *fd* in the file descriptor set *fdset*.

void *FD_ZERO*(fd_set **fdset*)
Initializes the file descriptor set *fdset* to have zero bits for all file descriptors.

If implemented as macros, these may evaluate their arguments more than once, so applications should ensure that the arguments they supply are never expressions with side effects.

The following shall be defined as a macro:

FD_SETSIZE
Maximum number of file descriptors in an **fd_set** structure.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int  pselect(int, fd_set *restrict, fd_set *restrict, fd_set *restrict,
         const struct timespec *restrict, const sigset_t *restrict);
int  select(int, fd_set *restrict, fd_set *restrict, fd_set *restrict,
         struct timeval *restrict);
```

Inclusion of the **<sys/select.h>** header may make visible all symbols from the headers **<signal.h>**, **<sys/time.h>**, and **<time.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<signal.h>, **<sys/time.h>**, **<sys/types.h>**, **<time.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *pselect*( ), *select*( )

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1g-2000.

The requirement for the **fd_set** structure to have a member *fds_bits* has been removed as per The Open Group Base Resolution bwg2001-005.

NAME

sys/sem.h — XSI semaphore facility

SYNOPSIS

XSI

```
#include <sys/sem.h>
```

DESCRIPTION

The **<sys/sem.h>** header shall define the following constants and structures.

Semaphore operation flags:

SEM_UNDO	Set up adjust on exit entry.

Command definitions for the *semctl*( ) function shall be provided as follows:

GETNCNT	Get *semncnt*.
GETPID	Get *sempid*.
GETVAL	Get *semval*.
GETALL	Get all cases of *semval*.
GETZCNT	Get *semzcnt*.
SETVAL	Set *semval*.
SETALL	Set all cases of *semval*.

The **semid_ds** structure shall contain the following members:

`struct ipc_perm`	`sem_perm`	Operation permission structure.
`unsigned short`	`sem_nsems`	Number of semaphores in set.
`time_t`	`sem_otime`	Last *semop*( ) time.
`time_t`	`sem_ctime`	Last time changed by *semctl*( ).

The **pid_t**, **time_t**, **key_t**, and **size_t** types shall be defined as described in **<sys/types.h>**.

A semaphore shall be represented by an anonymous structure containing the following members:

`unsigned short`	`semval`	Semaphore value.
`pid_t`	`sempid`	Process ID of last operation.
`unsigned short`	`semncnt`	Number of processes waiting for *semval* to become greater than current value.
`unsigned short`	`semzcnt`	Number of processes waiting for *semval* to become 0.

The **sembuf** structure shall contain the following members:

`unsigned short`	`sem_num`	Semaphore number.
`short`	`sem_op`	Semaphore operation.
`short`	`sem_flg`	Operation flags.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int   semctl(int, int, int, ...);
int   semget(key_t, int, int);
int   semop(int, struct sembuf *, size_t);
```

In addition, all of the symbols from **<sys/ipc.h>** shall be defined when this header is included.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, *semctl*( ), *semget*( ), *semop*( )

CHANGE HISTORY

First released in Issue 2. Derived from System V Release 2.0.

NAME

sys/shm.h — XSI shared memory facility

SYNOPSIS

XSI

```
#include <sys/shm.h>
```

DESCRIPTION

The **<sys/shm.h>** header shall define the following symbolic constants:

SHM_RDONLY	Attach read-only (else read-write).
SHM_RND	Round attach address to SHMLBA.

The **<sys/shm.h>** header shall define the following symbolic value:

SHMLBA	Segment low boundary address multiple.

The following data types shall be defined through **typedef**:

shmatt_t	Unsigned integer used for the number of current attaches that must be able to store values at least as large as a type **unsigned short**.

The **shmid_ds** structure shall contain the following members:

`struct ipc_perm`	`shm_perm`	Operation permission structure.
`size_t`	`shm_segsz`	Size of segment in bytes.
`pid_t`	`shm_lpid`	Process ID of last shared memory operation.
`pid_t`	`shm_cpid`	Process ID of creator.
`shmatt_t`	`shm_nattch`	Number of current attaches.
`time_t`	`shm_atime`	Time of last *shmat*( ).
`time_t`	`shm_dtime`	Time of last *shmdt*( ).
`time_t`	`shm_ctime`	Time of last change by *shmctl*( ).

The **pid_t**, **time_t**, **key_t**, and **size_t** types shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
void *shmat(int, const void *, int);
int   shmctl(int, int, struct shmid_ds *);
int   shmdt(const void *);
int   shmget(key_t, size_t, int);
```

In addition, all of the symbols from **<sys/ipc.h>** shall be defined when this header is included.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *shmat*( ), *shmctl*( ), *shmdt*( ), *shmget*( )

CHANGE HISTORY

First released in Issue 2. Derived from System V Release 2.0.

Issue 5

The type of *shm_segsz* is changed from **int** to **size_t**.

NAME

sys/socket.h — main sockets header

SYNOPSIS

```
#include <sys/socket.h>
```

DESCRIPTION

The **<sys/socket.h>** header shall define the type **socklen_t**, which is an integer type of width of at least 32 bits; see APPLICATION USAGE.

The **<sys/socket.h>** header shall define the unsigned integer type **sa_family_t**.

The **<sys/socket.h>** header shall define the **sockaddr** structure that includes at least the following members:

`sa_family_t`	`sa_family`	Address family.
`char`	`sa_data[]`	Socket address (variable-length data).

The **sockaddr** structure is used to define a socket address which is used in the *bind*( ), *connect*( ), *getpeername*( ), *getsockname*( ), *recvfrom*( ), and *sendto*( ) functions.

The **<sys/socket.h>** header shall define the **sockaddr_storage** structure. This structure shall be:

- Large enough to accommodate all supported protocol-specific address structures
- Aligned at an appropriate boundary so that pointers to it can be cast as pointers to protocol-specific address structures and used to access the fields of those structures without alignment problems

The **sockaddr_storage** structure shall contain at least the following members:

```
sa_family_t   ss_family
```

When a **sockaddr_storage** structure is cast as a **sockaddr** structure, the *ss_family* field of the **sockaddr_storage** structure shall map onto the *sa_family* field of the **sockaddr** structure. When a **sockaddr_storage** structure is cast as a protocol-specific address structure, the *ss_family* field shall map onto a field of that structure that is of type **sa_family_t** and that identifies the protocol's address family.

The **<sys/socket.h>** header shall define the **msghdr** structure that includes at least the following members:

`void`	`*msg_name`	Optional address.
`socklen_t`	`msg_namelen`	Size of address.
`struct iovec`	`*msg_iov`	Scatter/gather array.
`int`	`msg_iovlen`	Members in msg_iov.
`void`	`*msg_control`	Ancillary data; see below.
`socklen_t`	`msg_controllen`	Ancillary data buffer len.
`int`	`msg_flags`	Flags on received message.

The **msghdr** structure is used to minimize the number of directly supplied parameters to the *recvmsg*( ) and *sendmsg*( ) functions. This structure is used as a *value*-*result* parameter in the *recvmsg*( ) function and *value* only for the *sendmsg*( ) function.

The **iovec** structure shall be defined as described in **<sys/uio.h>**.

The **<sys/socket.h>** header shall define the **cmsghdr** structure that includes at least the following members:

`socklen_t`	`cmsg_len`	Data byte count, including the cmsghdr.
`int`	`cmsg_level`	Originating protocol.

`int`	`cmsg_type`	Protocol-specific type.

The **cmsghdr** structure is used for storage of ancillary data object information.

Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data.

The values for *cmsg_level* shall be legal values for the *level* argument to the *getsockopt*( ) and *setsockopt*( ) functions. The system documentation shall specify the *cmsg_type* definitions for the supported protocols.

Ancillary data is also possible at the socket level. The **<sys/socket.h>** header defines the following macro for use as the *cmsg_type* value when *cmsg_level* is SOL_SOCKET:

SCM_RIGHTS	Indicates that the data array contains the access rights to be sent or received.

The **<sys/socket.h>** header defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

CMSG_DATA(*cmsg*)

If the argument is a pointer to a **cmsghdr** structure, this macro shall return an unsigned character pointer to the data array associated with the **cmsghdr** structure.

CMSG_NXTHDR(*mhdr,cmsg*)

If the first argument is a pointer to a **msghdr** structure and the second argument is a pointer to a **cmsghdr** structure in the ancillary data pointed to by the *msg_control* field of that **msghdr** structure, this macro shall return a pointer to the next **cmsghdr** structure, or a null pointer if this structure is the last **cmsghdr** in the ancillary data.

CMSG_FIRSTHDR(*mhdr*)

If the argument is a pointer to a **msghdr** structure, this macro shall return a pointer to the first **cmsghdr** structure in the ancillary data associated with this **msghdr** structure, or a null pointer if there is no ancillary data associated with the **msghdr** structure.

The **<sys/socket.h>** header shall define the **linger** structure that includes at least the following members:

`int`	`l_onoff`	Indicates whether linger option is enabled.
`int`	`l_linger`	Linger time, in seconds.

The **<sys/socket.h>** header shall define the following macros, with distinct integer values:

	SOCK_DGRAM	Datagram socket.
RS	SOCK_RAW	Raw Protocol Interface.
	SOCK_SEQPACKET	Sequenced-packet socket.
	SOCK_STREAM	Byte-stream socket.

The **<sys/socket.h>** header shall define the following macro for use as the *level* argument of *setsockopt*( ) and *getsockopt*( ).

SOL_SOCKET	Options to be accessed at socket level, not protocol level.

The **<sys/socket.h>** header shall define the following macros, with distinct integer values, for use as the *option_name* argument in *getsockopt*( ) or *setsockopt*( ) calls:

SO_ACCEPTCONN	Socket is accepting connections.

SO_BROADCAST	Transmission of broadcast messages is supported.
SO_DEBUG	Debugging information is being recorded.
SO_DONTROUTE	Bypass normal routing.
SO_ERROR	Socket error status.
SO_KEEPALIVE	Connections are kept alive with periodic messages.
SO_LINGER	Socket lingers on close.
SO_OOBINLINE	Out-of-band data is transmitted in line.
SO_RCVBUF	Receive buffer size.
SO_RCVLOWAT	Receive ‘‘low water mark’’.
SO_RCVTIMEO	Receive timeout.
SO_REUSEADDR	Reuse of local addresses is supported.
SO_SNDBUF	Send buffer size.
SO_SNDLOWAT	Send ‘‘low water mark’’.
SO_SNDTIMEO	Send timeout.
SO_TYPE	Socket type.

The **<sys/socket.h>** header shall define the following macro as the maximum *backlog* queue length which may be specified by the *backlog* field of the *listen*( ) function:

SOMAXCONN	The maximum *backlog* queue length.

The **<sys/socket.h>** header shall define the following macros, with distinct integer values, for use as the valid values for the *msg_flags* field in the **msghdr** structure, or the *flags* parameter in *recvfrom*( ), *recvmsg*( ), *sendmsg*( ), or *sendto*( ) calls:

MSG_CTRUNC	Control data truncated.
MSG_DONTROUTE	Send without using routing tables.
MSG_EOR	Terminates a record (if supported by the protocol).
MSG_OOB	Out-of-band data.
MSG_PEEK	Leave received data in queue.
MSG_TRUNC	Normal data truncated.
MSG_WAITALL	Attempt to fill the read buffer.

The **<sys/socket.h>** header shall define the following macros, with distinct integer values:

AF_INET	Internet domain sockets for use with IPv4 addresses.
IP6 AF_INET6	Internet domain sockets for use with IPv6 addresses.
AF_UNIX	UNIX domain sockets.
AF_UNSPEC	Unspecified .

The **<sys/socket.h>** header shall define the following macros, with distinct integer values:

SHUT_RD	Disables further receive operations.

SHUT_RDWR	Disables further send and receive operations.
SHUT_WR	Disables further send operations.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int     accept(int, struct sockaddr *restrict, socklen_t *restrict);
int     bind(int, const struct sockaddr *, socklen_t);
int     connect(int, const struct sockaddr *, socklen_t);
int     getpeername(int, struct sockaddr *restrict, socklen_t *restrict);
int     getsockname(int, struct sockaddr *restrict, socklen_t *restrict);
int     getsockopt(int, int, int, void *restrict, socklen_t *restrict);
int     listen(int, int);
ssize_t recv(int, void *, size_t, int);
ssize_t recvfrom(int, void *restrict, size_t, int,
        struct sockaddr *restrict, socklen_t *restrict);
ssize_t recvmsg(int, struct msghdr *, int);
ssize_t send(int, const void *, size_t, int);
ssize_t sendmsg(int, const struct msghdr *, int);
ssize_t sendto(int, const void *, size_t, int, const struct sockaddr *,
        socklen_t);
int     setsockopt(int, int, int, const void *, socklen_t);
int     shutdown(int, int);
int     socket(int, int, int);
int     socketpair(int, int, int, int[2]);
```

Inclusion of **<sys/socket.h>** may also make visible all symbols from **<sys/uio.h>**.

APPLICATION USAGE

To forestall portability problems, it is recommended that applications not use values larger than $2^{31}-1$ for the **socklen_t** type.

The **sockaddr_storage** structure solves the problem of declaring storage for automatic variables which is both large enough and aligned enough for storing the socket address data structure of any family. For example, code with a file descriptor and without the context of the address family can pass a pointer to a variable of this type, where a pointer to a socket address structure is expected in calls such as *getpeername*( ), and determine the address family by accessing the received content after the call.

The example below illustrates a data structure which aligns on a 64-bit boundary. An implementation-defined field *_ss_align* following *_ss_pad1* is used to force a 64-bit alignment which covers proper alignment good enough for needs of at least **sockaddr_in6** (IPv6) and **sockaddr_in** (IPv4) address data structures. The size of padding field *_ss_pad1* depends on the chosen alignment boundary. The size of padding field *_ss_pad2* depends on the value of overall size chosen for the total size of the structure. This size and alignment are represented in the above example by implementation-defined (not required) constants _SS_MAXSIZE (chosen value 128) and _SS_ALIGNMENT (with chosen value 8). Constants _SS_PAD1SIZE (derived value 6) and _SS_PAD2SIZE (derived value 112) are also for illustration and not required. The implementation-defined definitions and structure field names above start with an underscore to denote implementation private name space. Portable code is not expected to access or reference those fields or constants.

```
/*
 *  Desired design of maximum size and alignment.
 */
```

```
#define _SS_MAXSIZE 128
    /* Implementation-defined maximum size. */
#define _SS_ALIGNSIZE (sizeof(int64_t))
    /* Implementation-defined desired alignment. */

/*
 *  Definitions used for sockaddr_storage structure paddings design.
 */
#define _SS_PAD1SIZE (_SS_ALIGNSIZE − sizeof(sa_family_t))
#define _SS_PAD2SIZE (_SS_MAXSIZE − (sizeof(sa_family_t)+
                      _SS_PAD1SIZE + _SS_ALIGNSIZE))
struct sockaddr_storage {
    sa_family_t  ss_family;  /* Address family. */
/*
 * Following fields are implementation-defined. */
 */
    char _ss_pad1[_SS_PAD1SIZE];
        /* 6-byte pad; this is to make implementation-defined
           pad up to alignment field that follows explicit in
           the data structure. */
    int64_t _ss_align;  /* Field to force desired structure
                           storage alignment. */
    char _ss_pad2[_SS_PAD2SIZE];
        /* 112-byte pad to achieve desired size,
           _SS_MAXSIZE value minus size of ss_family
           __ss_pad1, __ss_align fields is 112. */
};
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/uio.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *accept*( ), *bind*( ), *connect*( ), *getpeername*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), *socketpair*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the prototypes for *accept*( ), *getpeername*( ), *getsockname*( ), *getsockopt*( ), and *recvfrom*( ).

NAME

sys/stat.h — data returned by the stat( ) function

SYNOPSIS

```
#include <sys/stat.h>
```

DESCRIPTION

The **<sys/stat.h>** header shall define the structure of the data returned by the functions *fstat*( ), *lstat*( ), and *stat*( ).

The **stat** structure shall contain at least the following members:

	`dev_t`	`st_dev`	ID of device containing file.
	`ino_t`	`st_ino`	File serial number.
	`mode_t`	`st_mode`	Mode of file (see below).
	`nlink_t`	`st_nlink`	Number of hard links to the file.
	`uid_t`	`st_uid`	User ID of file.
	`gid_t`	`st_gid`	Group ID of file.
XSI	`dev_t`	`st_rdev`	Device ID (if file is character or block special).
	`off_t`	`st_size`	For regular files, the file size in bytes. For symbolic links, the length in bytes of the pathname contained in the symbolic link.
SHM			For a shared memory object, the length in bytes.
TYM			For a typed memory object, the length in bytes.
			For other file types, the use of this field is unspecified
	`time_t`	`st_atime`	Time of last access.
	`time_t`	`st_mtime`	Time of last data modification.
	`time_t`	`st_ctime`	Time of last status change.
XSI	`blksize_t`	`st_blksize`	A file system-specific preferred I/O block size for this object. In some file system types, this may vary from file to file.
	`blkcnt_t`	`st_blocks`	Number of blocks allocated for this object.

File serial number and device ID taken together uniquely identify the file within the system. The **blkcnt_t**, **blksize_t**, **dev_t**, **ino_t**, **mode_t**, **nlink_t**, **uid_t**, **gid_t**, **off_t**, and **time_t** types shall be defined as described in **<sys/types.h>**. Times shall be given in seconds since the Epoch.

Unless otherwise specified, the structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful values for all file types defined in IEEE Std 1003.1-200x.

For symbolic links, the *st_mode* member shall contain meaningful information, which can be used with the file type macros described below, that take a *mode* argument. The *st_size* member shall contain the length, in bytes, of the pathname contained in the symbolic link. File mode bits and the contents of the remaining members of the **stat** structure are unspecified. The value returned in the *st_size* field shall be the length of the contents of the symbolic link, and shall not count a trailing null if one is present.

The following symbolic names for the values of type *mode_t* shall also be defined.

File type:

XSI	S_IFMT	Type of file.	
		S_IFBLK	Block special.

	S_IFCHR	Character special.
	S_IFIFO	FIFO special.
	S_IFREG	Regular.
	S_IFDIR	Directory.
	S_IFLNK	Symbolic link.
	S_IFSOCK	Socket.

File mode bits:

S_IRWXU	Read, write, execute/search by owner.	
	S_IRUSR	Read permission, owner.
	S_IWUSR	Write permission, owner.
	S_IXUSR	Execute/search permission, owner.
S_IRWXG	Read, write, execute/search by group.	
	S_IRGRP	Read permission, group.
	S_IWGRP	Write permission, group.
	S_IXGRP	Execute/search permission, group.
S_IRWXO	Read, write, execute/search by others.	
	S_IROTH	Read permission, others.
	S_IWOTH	Write permission, others.
	S_IXOTH	Execute/search permission, others.
S_ISUID	Set-user-ID on execution.	
S_ISGID	Set-group-ID on execution.	
XSI S_ISVTX	On directories, restricted deletion flag.	

The bits defined by S_IRUSR, S_IWUSR, S_IXUSR, S_IRGRP, S_IWGRP, S_IXGRP, S_IROTH,
XSI S_IWOTH, S_IXOTH, S_ISUID, S_ISGID, and S_ISVTX shall be unique.

S_IRWXU is the bitwise-inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR.

S_IRWXG is the bitwise-inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP.

S_IRWXO is the bitwise-inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

Implementations may OR other implementation-defined bits into S_IRWXU, S_IRWXG, and S_IRWXO, but they shall not overlap any of the other bits defined in this volume of IEEE Std 1003.1-200x. The *file permission bits* are defined to be those corresponding to the bitwise-inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO.

The following macros shall be provided to test whether a file is of the specified type. The value *m* supplied to the macros is the value of *st_mode* from a **stat** structure. The macro shall evaluate to a non-zero value if the test is true; 0 if the test is false.

S_ISBLK(*m*)	Test for a block special file.
S_ISCHR(*m*)	Test for a character special file.

S_ISDIR(*m*)	Test for a directory.
S_ISFIFO(*m*)	Test for a pipe or FIFO special file.
S_ISREG(*m*)	Test for a regular file.
S_ISLNK(*m*)	Test for a symbolic link.
S_ISSOCK(*m*)	Test for a socket.

The implementation may implement message queues, semaphores, or shared memory objects as distinct file types. The following macros shall be provided to test whether a file is of the specified type. The value of the *buf* argument supplied to the macros is a pointer to a **stat** structure. The macro shall evaluate to a non-zero value if the specified object is implemented as a distinct file type and the specified file type is contained in the **stat** structure referenced by *buf*. Otherwise, the macro shall evaluate to zero.

S_TYPEISMQ(*buf*)	Test for a message queue.
S_TYPEISSEM(*buf*)	Test for a semaphore.
S_TYPEISSHM(*buf*)	Test for a shared memory object.

TYM The implementation may implement typed memory objects as distinct file types, and the following macro shall test whether a file is of the specified type. The value of the *buf* argument supplied to the macros is a pointer to a **stat** structure. The macro shall evaluate to a non-zero value if the specified object is implemented as a distinct file type and the specified file type is contained in the **stat** structure referenced by *buf*. Otherwise, the macro shall evaluate to zero.

S_TYPEISTMO(*buf*)	Test macro for a typed memory object.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int    chmod(const char *, mode_t);
int    fchmod(int, mode_t);
int    fstat(int, struct stat *);
int    lstat(const char *restrict, struct stat *restrict);
int    mkdir(const char *, mode_t);
int    mkfifo(const char *, mode_t);
int    mknod(const char *, mode_t, dev_t);
int    stat(const char *restrict, struct stat *restrict);
mode_t umask(mode_t);
```

XSI (mknod)

APPLICATION USAGE

Use of the macros is recommended for determining the type of a file.

RATIONALE

A conforming C-language application must include **<sys/stat.h>** for functions that have arguments or return values of type **mode_t**, so that symbolic values for that type can be used. An alternative would be to require that these constants are also defined by including **<sys/types.h>**.

The S_ISUID and S_ISGID bits may be cleared on any write, not just on *open*( ), as some historical implementations do it.

System calls that update the time entry fields in the **stat** structure must be documented by the implementors. POSIX-conforming systems should not update the time entry fields for functions listed in the System Interfaces volume of IEEE Std 1003.1-200x unless the standard requires that they do, except in the case of documented extensions to the standard.

Note that *st_dev* must be unique within a Local Area Network (LAN) in a ‘‘system’’ made up of multiple computers’ file systems connected by a LAN.

Networked implementations of a POSIX-conforming system must guarantee that all files visible within the file tree (including parts of the tree that may be remotely mounted from other machines on the network) on each individual processor are uniquely identified by the combination of the *st_ino* and *st_dev* fields.

FUTURE DIRECTIONS

No new S_IFMT symbolic names for the file type values of **mode_t** will be defined by IEEE Std 1003.1-200x; if new file types are required, they will only be testable through *S_ISxx*( ) macros instead.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *chmod*( ), *fchmod*( ), *fstat*( ), *lstat*( ), *mkdir*( ), *mkfifo*( ), *mknod*( ), *stat*( ), *umask*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with POSIX Realtime Extension.

The type of *st_blksize* is changed from **long** to **blksize_t**; the type of *st_blocks* is changed from **long** to **blkcnt_t**.

Issue 6

The S_TYPEISMQ( ), S_TYPEISSEM( ), and S_TYPEISSHM( ) macros are unconditionally mandated.

The Open Group Corrigendum U035/4 is applied. In the DESCRIPTION, the types **blksize_t** and **blkcnt_t** have been described.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **dev_t**, **ino_t**, **mode_t**, **nlink_t**, **uid_t**, **gid_t**, **off_t**, and **time_t** types are mandated.

S_IFSOCK and S_ISSOCK are added for sockets.

The description of **stat** structure members is changed to reflect contents when file type is a symbolic link.

The test macro S_TYPEISTMO is added for alignment with IEEE Std 1003.1j-2000.

The **restrict** keyword is added to the prototypes for *lstat*( ) and *stat*( ).

The *lstat*( ) function is now mandatory.

NAME

sys/statvfs.h — VFS File System information structure

SYNOPSIS

XSI

```
#include <sys/statvfs.h>
```

DESCRIPTION

The **<sys/statvfs.h>** header shall define the **statvfs** structure that includes at least the following members:

`unsigned long`	`f_bsize`	File system block size.
`unsigned long`	`f_frsize`	Fundamental file system block size.
`fsblkcnt_t`	`f_blocks`	Total number of blocks on file system in units of *f_frsize*.
`fsblkcnt_t`	`f_bfree`	Total number of free blocks.
`fsblkcnt_t`	`f_bavail`	Number of free blocks available to non-privileged process.
`fsfilcnt_t`	`f_files`	Total number of file serial numbers.
`fsfilcnt_t`	`f_ffree`	Total number of free file serial numbers.
`fsfilcnt_t`	`f_favail`	Number of file serial numbers available to non-privileged process.
`unsigned long`	`f_fsid`	File system ID.
`unsigned long`	`f_flag`	Bit mask of *f_flag* values.
`unsigned long`	`f_namemax`	Maximum filename length.

The **fsblkcnt_t** and **fsfilcnt_t** types shall be defined as described in **<sys/types.h>**.

The following flags for the *f_flag* member shall be defined:

`ST_RDONLY`	Read-only file system.
`ST_NOSUID`	Does not support setuid/setgid semantics.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int statvfs(const char *restrict, struct statvfs *restrict);
int fstatvfs(int, struct statvfs *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *fstatvfs*( ), *statvfs*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

The type of *f_blocks*, *f_bfree*, and *f_bavail* is changed from **unsigned long** to **fsblkcnt_t**; the type of *f_files*, *f_ffree*, and *f_favail* is changed from **unsigned long** to **fsfilcnt_t**.

Issue 6

The Open Group Corrigendum U035/5 is applied. In the DESCRIPTION, the types **fsblkcnt_t** and **fsfilcnt_t** have been described.

The **restrict** keyword is added to the prototype for *statvfs*( ).

NAME

sys/time.h — time types

SYNOPSIS

XSI
```
#include <sys/time.h>
```

DESCRIPTION

The **<sys/time.h>** header shall define the **timeval** structure that includes at least the following members:

`time_t`	`tv_sec`	Seconds.
`suseconds_t`	`tv_usec`	Microseconds.

The **<sys/time.h>** header shall define the **itimerval** structure that includes at least the following members:

`struct timeval`	`it_interval`	Timer interval.
`struct timeval`	`it_value`	Current value.

The **time_t** and **suseconds_t** types shall be defined as described in **<sys/types.h>**.

The **fd_set** type shall be defined as described in **<sys/select.h>**.

The **<sys/time.h>** header shall define the following values for the *which* argument of *getitimer*( ) and *setitimer*( ):

ITIMER_REAL	Decrements in real time.
ITIMER_VIRTUAL	Decrements in process virtual time.
ITIMER_PROF	Decrements both in process virtual time and when the system is running on behalf of the process.

The following shall be defined as described in **<sys/select.h>**:

FD_CLR( )
FD_ISSET( )
FD_SET( )
FD_ZERO( )
FD_SETSIZE( )

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int   getitimer(int, struct itimerval *);
int   gettimeofday(struct timeval *restrict, void *restrict);
int   select(int, fd_set *restrict, fd_set *restrict, fd_set *restrict,
          struct timeval *restrict);
int   setitimer(int, const struct itimerval *restrict,
          struct itimerval *restrict);
int   utimes(const char *, const struct timeval [2]); (LEGACY)
```

Inclusion of the **<sys/time.h>** header may make visible all symbols from the **<sys/select.h>** header.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *getitimer*( ), *gettimeofday*( ), *select*( ), *setitimer*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

The type of *tv_usec* is changed from **long** to **suseconds_t**.

Issue 6

The **restrict** keyword is added to the prototypes for *gettimeofday*( ), *select*( ), and *setitimer*( ).

The note is added that inclusion of this header may also make symbols visible from **<sys/socket.h>**.

The *utimes*( ) function is marked LEGACY.

NAME

sys/timeb.h — additional definitions for date and time

SYNOPSIS

XSI
```
#include <sys/timeb.h>
```

DESCRIPTION

The **<sys/timeb.h>** header shall define the **timeb** structure that includes at least the following members:

`time_t`	`time`	The seconds portion of the current time.
`unsigned short`	`millitm`	The milliseconds portion of the current time.
`short`	`timezone`	The local timezone in minutes west of Greenwich.
`short`	`dstflag`	TRUE if Daylight Savings Time is in effect.

The **time_t** type shall be defined as described in **<sys/types.h>**.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided. |

```
int   ftime(struct timeb *); (LEGACY)
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, **<time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

The *ftime*( ) function is marked LEGACY.

NAME

sys/times.h — file access and modification times structure

SYNOPSIS

```
#include <sys/times.h>
```

DESCRIPTION

The **<sys/times.h>** header shall define the structure **tms**, which is returned by *times*() and includes at least the following members:

```
clock_t  tms_utime   User CPU time.
clock_t  tms_stime   System CPU time.
clock_t  tms_cutime  User CPU time of terminated child processes.
clock_t  tms_cstime  System CPU time of terminated child processes.
```

The **clock_t** type shall be defined as described in **<sys/types.h>**.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided. |

```
clock_t times(struct tms *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *times*()

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

sys/types.h — data types

SYNOPSIS

```
#include <sys/types.h>
```

DESCRIPTION

The **<sys/types.h>** header shall include definitions for at least the following types:

	blkcnt_t	Used for file block counts.
	blksize_t	Used for block sizes.
XSI	**clock_t**	Used for system times in clock ticks or CLOCKS_PER_SEC; see **<time.h>**.
TMR	**clockid_t**	Used for clock ID type in the clock and timer functions.
	dev_t	Used for device IDs.
XSI	**fsblkcnt_t**	Used for file system block counts.
XSI	**fsfilcnt_t**	Used for file system file counts.
	gid_t	Used for group IDs.
XSI	**id_t**	Used as a general identifier; can be used to contain at least a **pid_t**, **uid_t**, or **gid_t**.
	ino_t	Used for file serial numbers.
XSI	**key_t**	Used for XSI interprocess communication.
	mode_t	Used for some file attributes.
	nlink_t	Used for link counts.
	off_t	Used for file sizes.
	pid_t	Used for process IDs and process group IDs.
THR	**pthread_attr_t**	Used to identify a thread attribute object.
BAR	**pthread_barrier_t**	Used to identify a barrier.
BAR	**pthread_barrierattr_t**	Used to define a barrier attributes object.
THR	**pthread_cond_t**	Used for condition variables.
THR	**pthread_condattr_t**	Used to identify a condition attribute object.
THR	**pthread_key_t**	Used for thread-specific data keys.
THR	**pthread_mutex_t**	Used for mutexes.
THR	**pthread_mutexattr_t**	Used to identify a mutex attribute object.
THR	**pthread_once_t**	Used for dynamic package initialization.
THR	**pthread_rwlock_t**	Used for read-write locks.
THR	**pthread_rwlockattr_t**	Used for read-write lock attributes.
SPI	**pthread_spinlock_t**	Used to identify a spin lock.
THR	**pthread_t**	Used to identify a thread.

	size_t	Used for sizes of objects.
	ssize_t	Used for a count of bytes or an error indication.
XSI	**suseconds_t**	Used for time in microseconds
	time_t	Used for time in seconds.
TMR	**timer_t**	Used for timer ID returned by *timer_create*( ).
	uid_t	Used for user IDs.
XSI	**useconds_t**	Used for time in microseconds.

All of the types shall be defined as arithmetic types of an appropriate length, with the following exceptions:

XSI **key_t**
THR **pthread_attr_t**
BAR **pthread_barrier_t**
pthread_barrierattr_t
THR **pthread_cond_t**
pthread_condattr_t
pthread_key_t
pthread_mutex_t
pthread_mutexattr_t
pthread_once_t
pthread_rwlock_t
pthread_rwlockattr_t
SPI **pthread_spinlock_t**
TRC **trace_attr_t**
trace_event_id_t
TRC TEF **trace_event_set_t**
TRC **trace_id_t**

Additionally:

- **mode_t** shall be an integer type.
- **nlink_t**, **uid_t**, **gid_t**, and **id_t** shall be integer types.
- **blkcnt_t** and **off_t** shall be signed integer types.
- XSI **fsblkcnt_t**, **fsfilcnt_t**, and **ino_t** shall be defined as unsigned integer types.
- **size_t** shall be an unsigned integer type.
- **blksize_t**, **pid_t**, and **ssize_t** shall be signed integer types.
- **time_t** and **clock_t** shall be integer or real-floating types.

XSI The type **ssize_t** shall be capable of storing values at least in the range [–1, {SSIZE_MAX}]. The type **useconds_t** shall be an unsigned integer type capable of storing values at least in the range [0, 1 000 000]. The type **suseconds_t** shall be a signed integer type capable of storing values at least in the range [–1, 1 000 000].

The implementation shall support one or more programming environments in which the widths of **blksize_t**, **pid_t**, **size_t**, **ssize_t**, **suseconds_t**, and **useconds_t** are no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

There are no defined comparison or assignment operators for the following types: |

THR **pthread_attr_t**
BAR **pthread_barrier_t**
pthread_barrierattr_t
THR **pthread_cond_t**
pthread_condattr_t
pthread_mutex_t
pthread_mutexattr_t
pthread_rwlock_t
pthread_rwlockattr_t
SPI **pthread_spinlock_t**
TRC **trace_attr_t**

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<time.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *confstr*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf* |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The **clockid_t** and **timer_t** types are defined for alignment with the POSIX Realtime Extension.

The types **blkcnt_t**, **blksize_t**, **fsblkcnt_t**, **fsfilcnt_t**, and **suseconds_t** are added.

Large File System extensions are added.

Updated for alignment with the POSIX Threads Extension.

Issue 6

The **pthread_barrier_t**, **pthread_barrierattr_t**, and **pthread_spinlock_t** types are added for alignment with IEEE Std 1003.1j-2000.

The margin code is changed from XSI to THR for the **pthread_rwlock_t** and **pthread_rwlockattr_t** types as Read-Write Locks have been absorbed into the POSIX Threads option. The threads types are now marked THR.

NAME

sys/uio.h — definitions for vector I/O operations

SYNOPSIS

XSI

```
#include <sys/uio.h>
```

DESCRIPTION

The **<sys/uio.h>** header shall define the **iovec** structure that includes at least the following members:

`void`	`*iov_base`	Base address of a memory region for input or output.
`size_t`	`iov_len`	The size of the memory pointed to by *iov_base*.

The **<sys/uio.h>** header uses the **iovec** structure for scatter/gather I/O.

The **ssize_t** and **size_t** types shall be defined as described in **<sys/types.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
ssize_t readv(int, const struct iovec *, int);
ssize_t writev(int, const struct iovec *, int);
```

APPLICATION USAGE

The implementation can put a limit on the number of scatter/gather elements which can be processed in one call. The symbol {IOV_MAX} defined in **<limits.h>** should always be used to learn about the limits instead of assuming a fixed value.

RATIONALE

Traditionally, the maximum number of scatter/gather elements the system can process in one call were described by the symbolic value {UIO_MAXIOV}. In IEEE Std 1003.1-200x this value was replaced by the constant {IOV_MAX} which can be found in **<limits.h>**.

FUTURE DIRECTIONS

None.

SEE ALSO

<limits.h>, **<sys/types.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *read*( ), *write*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

Text referring to scatter/gather I/O is added to the DESCRIPTION.

NAME

sys/un.h — definitions for UNIX domain sockets

SYNOPSIS

```
#include <sys/un.h>
```

DESCRIPTION

The **<sys/un.h>** header shall define the **sockaddr_un** structure that includes at least the following members:

`sa_family_t`	`sun_family`	Address family.
`char`	`sun_path[]`	Socket pathname.

The **sockaddr_un** structure is used to store addresses for UNIX domain sockets. Values of this type shall be cast by applications to **struct sockaddr** for use with socket functions.

The **sa_family_t** type shall be defined as described in **<sys/socket.h>**.

APPLICATION USAGE

The size of *sun_path* has intentionally been left undefined. This is because different implementations use different sizes. For example, BSD4.3 uses a size of 108, and BSD4.4 uses a size of 104. Since most implementations originate from BSD versions, the size is typically in the range 92 to 108.

Applications should not assume a particular length for *sun_path* or assume that it can hold _POSIX_PATH_MAX characters (255).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/socket.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *bind*( ), *socket*( ), *socketpair*( )

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

sys/utsname.h — system name structure

SYNOPSIS

```
#include <sys/utsname.h>
```

DESCRIPTION

The **<sys/utsname.h>** header shall define the structure **utsname** which shall include at least the following members:

`char`	`sysname[]`	Name of this implementation of the operating system.
`char`	`nodename[]`	Name of this node within an implementation-defined communications network.
`char`	`release[]`	Current release level of this implementation.
`char`	`version[]`	Current version level of this release.
`char`	`machine[]`	Name of the hardware type on which the system is running.

The character arrays are of unspecified size, but the data stored in them shall be terminated by a null byte.

The following shall be declared as a function and may also be defined as a macro:

```
int uname(struct utsname *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *uname*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

sys/wait.h — declarations for waiting

SYNOPSIS

```
#include <sys/wait.h>
```

DESCRIPTION

The **<sys/wait.h>** header shall define the following symbolic constants for use with *waitpid*( ):

WNOHANG	Do not hang if no status is available; return immediately.
WUNTRACED	Report status of stopped child process.

The **<sys/wait.h>** header shall define the following macros for analysis of process status values:

WEXITSTATUS	Return exit status.
XSI WIFCONTINUED	True if child has been continued
WIFEXITED	True if child exited normally.
WIFSIGNALED	True if child exited due to uncaught signal.
WIFSTOPPED	True if child is currently stopped.
WSTOPSIG	Return signal number that caused process to stop.
WTERMSIG	Return signal number that caused process to terminate.

XSI The following symbolic constants shall be defined as possible values for the *options* argument to *waitid*( ):

WEXITED	Wait for processes that have exited.
WSTOPPED	Status is returned for any child that has stopped upon receipt of a signal.
WCONTINUED	Status is returned for any child that was stopped and has been continued.
WNOHANG	Return immediately if there are no children to wait for.
WNOWAIT	Keep the process whose status is returned in *infop* in a waitable state.

The type **idtype_t** shall be defined as an enumeration type whose possible values shall include at least the following:

P_ALL
P_PID
P_PGID

The **id_t** and **pid_t** types shall be defined as described in **<sys/types.h>**.

XSI The **siginfo_t** type shall be defined as described in **<signal.h>**.

The **rusage** structure shall be defined as described in **<sys/resource.h>**.

Inclusion of the **<sys/wait.h>** header may also make visible all symbols from **<signal.h>** and **<sys/resource.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
pid_t  wait(int *);
int    waitid(idtype_t, id_t, siginfo_t *, int);
pid_t  waitpid(pid_t, int *, int);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<signal.h>, **<sys/resource.h>**, **<sys/types.h>**, **<sys/wait.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *wait*( ), *waitid*( )

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The *wait3*( ) function is removed.

NAME

syslog — definitions for system error logging

SYNOPSIS

XSI

```
#include <syslog.h>
```

DESCRIPTION

The **<syslog.h>** header shall define the following symbolic constants, zero or more of which may be OR'ed together to form the *logopt* option of *openlog*( ):

LOG_PID	Log the process ID with each message.
LOG_CONS	Log to the system console on error.
LOG_NDELAY	Connect to syslog daemon immediately.
LOG_ODELAY	Delay open until *syslog*( ) is called.
LOG_NOWAIT	Do not wait for child processes.

The following symbolic constants shall be defined as possible values of the *facility* argument to *openlog*( ):

LOG_KERN	Reserved for message generated by the system.
LOG_USER	Message generated by a process.
LOG_MAIL	Reserved for message generated by mail system.
LOG_NEWS	Reserved for message generated by news system.
LOG_UUCP	Reserved for message generated by UUCP system.
LOG_DAEMON	Reserved for message generated by system daemon.
LOG_AUTH	Reserved for message generated by authorization daemon.
LOG_CRON	Reserved for message generated by the clock daemon.
LOG_LPR	Reserved for message generated by printer system.
LOG_LOCAL0	Reserved for local use.
LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.
LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

The following shall be declared as macros for constructing the *maskpri* argument to *setlogmask*( ). The following macros expand to an expression of type **int** when the argument *pri* is an expression of type **int**:

LOG_MASK(*pri*)	A mask for priority *pri*.

The following constants shall be defined as possible values for the *priority* argument of *syslog*( ):

LOG_EMERG	A panic condition was reported to all processes.
LOG_ALERT	A condition that should be corrected immediately.
LOG_CRIT	A critical condition.
LOG_ERR	An error message.
LOG_WARNING	A warning message.
LOG_NOTICE	A condition requiring special handling.
LOG_INFO	A general information message.
LOG_DEBUG	A message useful for debugging programs.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
void  closelog(void);
void  openlog(const char *, int, int);
int   setlogmask(int);
void  syslog(int, const char *, ...);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *closelog*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved to X/Open UNIX to BASE.

NAME

tar.h — extended tar definitions

SYNOPSIS

```
#include <tar.h>
```

DESCRIPTION

The **<tar.h>** header shall define header block definitions as follows.

General definitions:

Name	Description	Value
TMAGIC	`"ustar"`	ustar plus null byte.
TMAGLEN	`6`	Length of the above.
TVERSION	`"00"`	00 without a null byte.
TVERSLEN	`2`	Length of the above.

Typeflag field definitions:

Name	Description	Value
REGTYPE	`'0'`	Regular file.
AREGTYPE	`'\0'`	Regular file.
LNKTYPE	`'1'`	Link.
SYMTYPE	`'2'`	Symbolic link.
CHRTYPE	`'3'`	Character special.
BLKTYPE	`'4'`	Block special.
DIRTYPE	`'5'`	Directory.
FIFOTYPE	`'6'`	FIFO special.
CONTTYPE	`'7'`	Reserved.

Mode field bit definitions (octal):

Name	Description	Value
TSUID	04000	Set UID on execution.
TSGID	02000	Set GID on execution.
TSVTX	01000	On directories, restricted deletion flag.
TUREAD	00400	Read by owner.
TUWRITE	00200	Write by owner special.
TUEXEC	00100	Execute/search by owner.
TGREAD	00040	Read by group.
TGWRITE	00020	Write by group.
TGEXEC	00010	Execute/search by group.
TOREAD	00004	Read by other.
TOWRITE	00002	Write by other.
TOEXEC	00001	Execute/search by other.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Shell and Utilities volume of IEEE Std 1003.1-200x, *pax*

CHANGE HISTORY

First released in Issue 3. Derived from the entry in the POSIX.1-1988 standard.

Issue 6

The SEE ALSO section now refers to *pax* since the Shell and Utilities volume of IEEE Std 1003.1-200x no longer contains the *tar* utility.

NAME

termios.h — define values for termios

SYNOPSIS

```
#include <termios.h>
```

DESCRIPTION

The **<termios.h>** header contains the definitions used by the terminal I/O interfaces (see Chapter 11 (on page 183) for the structures and names defined).

The termios Structure

The following data types shall be defined through **typedef**:

cc_t	Used for terminal special characters.
speed_t	Used for terminal baud rates.
tcflag_t	Used for terminal modes.

The above types shall be all unsigned integer types.

The implementation shall support one or more programming environments in which the widths of **cc_t**, **speed_t**, and **tcflag_t** are no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

The **termios** structure shall be defined, and shall include at least the following members:

`tcflag_t`	`c_iflag`	Input modes.
`tcflag_t`	`c_oflag`	Output modes.
`tcflag_t`	`c_cflag`	Control modes.
`tcflag_t`	`c_lflag`	Local modes.
`cc_t`	`c_cc[NCCS]`	Control characters.

A definition shall be provided for:

NCCS	Size of the array *c_cc* for control characters.

The following subscript names for the array *c_cc* shall be defined:

Subscript Usage		
Canonical Mode	**Non-Canonical Mode**	**Description**
VEOF		EOF character.
VEOL		EOL character.
VERASE		ERASE character.
VINTR	VINTR	INTR character.
VKILL		KILL character.
	VMIN	MIN value.
VQUIT	VQUIT	QUIT character.
VSTART	VSTART	START character.
VSTOP	VSTOP	STOP character.
VSUSP	VSUSP	SUSP character.
	VTIME	TIME value.

The subscript values shall be unique, except that the VMIN and VTIME subscripts may have the same values as the VEOF and VEOL subscripts, respectively.

The following flags shall be provided.

Input Modes

The *c_iflag* field describes the basic terminal input control:

	BRKINT	Signal interrupt on break.
	ICRNL	Map CR to NL on input.
	IGNBRK	Ignore break condition.
	IGNCR	Ignore CR.
	IGNPAR	Ignore characters with parity errors.
	INLCR	Map NL to CR on input.
	INPCK	Enable input parity check.
	ISTRIP	Strip character.
XSI	IXANY	Enable any character to restart output.
	IXOFF	Enable start/stop input control.
	IXON	Enable start/stop output control.
	PARMRK	Mark parity errors.

Output Modes

The *c_oflag* field specifies the system treatment of output:

	OPOST	Post-process output.	
XSI	ONLCR	Map NL to CR-NL on output.	
	OCRNL	Map CR to NL on output.	
	ONOCR	No CR output at column 0.	
	ONLRET	NL performs CR function.	
	OFILL	Use fill characters for delay.	
	NLDLY	Select newline delays:	
		NL0	<newline> type 0.
		NL1	<newline> type 1.
	CRDLY	Select carriage-return delays:	
		CR0	Carriage-return delay type 0.
		CR1	Carriage-return delay type 1.
		CR2	Carriage-return delay type 2.
		CR3	Carriage-return delay type 3.
	TABDLY	Select horizontal-tab delays:	
		TAB0	Horizontal-tab delay type 0.
		TAB1	Horizontal-tab delay type 1.
		TAB2	Horizontal-tab delay type 2.

	TAB3	Expand tabs to spaces.
BSDLY	Select backspace delays:	
	BS0	Backspace-delay type 0.
	BS1	Backspace-delay type 1.
VTDLY	Select vertical-tab delays:	
	VT0	Vertical-tab delay type 0.
	VT1	Vertical-tab delay type 1.
FFDLY	Select form-feed delays:	
	FF0	Form-feed delay type 0.
	FF1	Form-feed delay type 1.

Baud Rate Selection

The input and output baud rates are stored in the **termios** structure. These are the valid values for objects of type **speed_t**. The following values shall be defined, but not all baud rates need be supported by the underlying hardware.

B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

Control Modes

The *c_cflag* field describes the hardware control of the terminal; not all values specified are required to be supported by the underlying hardware:

CSIZE	Character size:	
	CS5	5 bits
	CS6	6 bits
	CS7	7 bits
	CS8	8 bits
CSTOPB	Send two stop bits, else one.	
CREAD	Enable receiver.	
PARENB	Parity enable.	
PARODD	Odd parity, else even.	
HUPCL	Hang up on last close.	
CLOCAL	Ignore modem status lines.	

The implementation shall support the functionality associated with the symbols CS7, CS8, CSTOPB, PARODD, and PARENB. |

Local Modes

The *c_lflag* field of the argument structure is used to control various terminal functions:

ECHO	Enable echo.
ECHOE	Echo erase character as error-correcting backspace.
ECHOK	Echo KILL.
ECHONL	Echo NL.
ICANON	Canonical input (erase and kill processing).
IEXTEN	Enable extended input character processing.
ISIG	Enable signals.
NOFLSH	Disable flush after interrupt or quit.
TOSTOP	Send SIGTTOU for background output.

Attribute Selection

The following symbolic constants for use with *tcsetattr*( ) are defined:

TCSANOW	Change attributes immediately.
TCSADRAIN	Change attributes when output has drained.
TCSAFLUSH	Change attributes when output has drained; also flush pending input.

Line Control

The following symbolic constants for use with *tcflush*( ) shall be defined:

TCIFLUSH	Flush pending input. Flush untransmitted output.
TCIOFLUSH	Flush both pending input and untransmitted output.
TCOFLUSH	Flush untransmitted output.

The following symbolic constants for use with *tcflow*( ) shall be defined:

TCIOFF	Transmit a STOP character, intended to suspend input data.
TCION	Transmit a START character, intended to restart input data.
TCOOFF	Suspend output.
TCOON	Restart output.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
speed_t cfgetispeed(const struct termios *);
speed_t cfgetospeed(const struct termios *);
int     cfsetispeed(struct termios *, speed_t);
int     cfsetospeed(struct termios *, speed_t);
int     tcdrain(int);
int     tcflow(int, int);
int     tcflush(int, int);
int     tcgetattr(int, struct termios *);
pid_t   tcgetsid(int);
int     tcsendbreak(int, int);
int     tcsetattr(int, int, const struct termios *);
```

APPLICATION USAGE

The following names are reserved for XSI-conformant systems to use as an extension to the above; therefore strictly conforming applications shall not use them:

CBAUD
DEFECHO
ECHOCTL
ECHOKE
ECHOPRT
EXTA
EXTB
FLUSHO
LOBLK
PENDIN
SWTCH
VDISCARD
VDSUSP
VLNEXT
VREPRINT
VSTATUS
VWERASE

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *cfgetispeed*( ), *cfgetospeed*( ), *cfsetispeed*( ), *cfsetospeed*( ), *getconf*( ), *tcdrain*( ), *tcflow*( ), *tcflush*( ), *tcgetattr*( ), *tcgetsid*( ), *tcsendbreak*( ), *tcsetattr*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*, Chapter 11 (on page 183)

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ISO POSIX-1 standard.

Issue 6

The LEGACY symbols IUCLC, ULCUC, and XCASE are removed.

FIPS 151-2 requirements for the symbols CS7, CS8, CSTOPB, PARODD, and PARENB are reaffirmed.

NAME

tgmath.h — type-generic macros

SYNOPSIS

```
#include <tgmath.h>
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The **<tgmath.h>** header shall include the headers **<math.h>** and **<complex.h>** and shall define several type-generic macros.

Of the functions contained within the **<math.h>** and **<complex.h>** headers without an *f* (**float**) or *l* (**long double**) suffix, several have one or more parameters whose corresponding real type is **double**. For each such function, except *modf*( ), there shall be a corresponding type-generic macro. The parameters whose corresponding real type is **double** in the function synopsis are generic parameters. Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.

Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

- First, if any argument for generic parameters has type **long double**, the type determined is **long double**.
- Otherwise, if any argument for generic parameters has type **double** or is of integer type, the type determined is **double**.
- Otherwise, the type determined is **float**.

For each unsuffixed function in the **<math.h>** header for which there is a function in the **<complex.h>** header with the same name except for a *c* prefix, the corresponding type-generic macro (for both functions) has the same name as the function in the **<math.h>** header. The corresponding type-generic macro for *fabs*( ) and *cabs*( ) is *fabs*( ).

<math.h> Function	**<complex.h> Function**	**Type-Generic Macro**
acos( )	*cacos*( )	*acos*( )
asin( )	*casin*( )	*asin*( )
atan( )	*catan*( )	*atan*( )
acosh( )	*cacosh*( )	*acosh*( )
asinh( )	*casinh*( )	*asinh*( )
atanh( )	*catanh*( )	*atanh*( )
cos( )	*ccos*( )	*cos*( )
sin( )	*csin*( )	*sin*( )
tan( )	*ctan*( )	*tan*( )
cosh( )	*ccosh*( )	*cosh*( )
sinh( )	*csinh*( )	*sinh*( )
tanh( )	*ctanh*( )	*tanh*( )
exp( )	*cexp*( )	*exp*( )
log( )	*clog*( )	*log*( )
pow( )	*cpow*( )	*pow*( )
sqrt( )	*csqrt*( )	*sqrt*( )
fabs( )	*cabs*( )	*fabs*( )

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

For each unsuffixed function in the **<math.h>** header without a *c*-prefixed counterpart in the **<complex.h>** header, the corresponding type-generic macro has the same name as the function. These type-generic macros are:

atan2( )	*fma*( )	*llround*( )	*remainder*( )
cbrt( )	*fmax*( )	*log10*( )	*remquo*( )
ceil( )	*fmin*( )	*log1p*( )	*rint*( )
copysign( )	*fmod*( )	*log2*( )	*round*( )
erf( )	*frexp*( )	*logb*( )	*scalbn*( )
erfc( )	*hypot*( )	*lrint*( )	*scalbln*( )
exp2( )	*ilogb*( )	*lround*( )	*tgamma*( )
expm1( )	*ldexp*( )	*nearbyint*( )	*trunc*( )
fdim( )	*lgamma*( )	*nextafter*( )	
floor( )	*llrint*( )	*nexttoward*( )	

If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.

For each unsuffixed function in the **<complex.h>** header that is not a *c*-prefixed counterpart to a function in the **<math.h>** header, the corresponding type-generic macro has the same name as the function. These type-generic macros are:

carg( )
cimag( )
conj( )
cproj( )
creal( )

Use of the macro with any real or complex argument invokes a complex function.

APPLICATION USAGE

With the declarations:

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

functions invoked by use of type-generic macros are shown in the following table:

Macro	Use Invokes
exp(*n*)	*exp*(*n*), the function
acosh(*f*)	*acoshf*(*f*)
sin(*d*)	*sin*(*d*), the function
atan(*ld*)	*atanl*(*ld*)

Macro	Use Invokes
log(*fc*)	*clogf*(*fc*)
sqrt(*dc*)	*csqrt*(*dc*)
pow(*ldc*,*f*)	*cpowl*(*ldc*, *f*)
remainder(*n*,*n*)	*remainder*(*n*, *n*), the function
nextafter(*d*,*f*)	*nextafter*(*d*, *f*), the function
nexttoward(*f*,*ld*)	*nexttowardf*(*f*, *ld*)
copysign(*n*,*ld*)	*copysignl*(*n*, *ld*)
ceil(*fc*)	Undefined behavior
rint(*dc*)	Undefined behavior
fmax(*ldc*,*ld*)	Undefined behavior
carg(*n*)	*carg*(*n*), the function
cproj(*f*)	*cprojf*(*f*)
creal(*d*)	*creal*(*d*), the function
cimag(*ld*)	*cimagl*(*ld*)
cabs(*fc*)	*cabsf*(*fc*)
carg(*dc*)	*carg*(*dc*), the function
cproj(*ldc*)	*cprojl*(*ldc*)

RATIONALE

Type-generic macros allow calling a function whose type is determined by the argument type, as is the case for C operators such as `'+'` and `'*'`. For example, with a type-generic *cos*( ) macro, the expression *cos*((**float**)*x*) will have type **float**. This feature enables writing more portably efficient code and alleviates need for awkward casting and suffixing in the process of porting or adjusting precision. Generic math functions are a widely appreciated feature of Fortran.

The only arguments that affect the type resolution are the arguments corresponding to the parameters that have type **double** in the synopsis. Hence the type of a type-generic call to *nexttoward*( ), whose second parameter is **long double** in the synopsis, is determined solely by the type of the first argument.

The term ‘‘type-generic’’ was chosen over the proposed alternatives of intrinsic and overloading. The term is more specific than intrinsic, which already is widely used with a more general meaning, and reflects a closer match to Fortran’s generic functions than to C++ overloading.

The macros are placed in their own header in order not to silently break old programs that include the **<math.h>** header; for example, with:

```
printf ("%e", sin(x))
```

modf(**double**, **double** *) is excluded because no way was seen to make it safe without complicating the type resolution.

The implementation might, as an extension, endow appropriate ones of the macros that IEEE Std 1003.1-200x specifies only for real arguments with the ability to invoke the complex functions.

IEEE Std 1003.1-200x does not prescribe any particular implementation mechanism for generic macros. It could be implemented simply with built-in macros. The generic macro for *sqrt*( ), for example, could be implemented with:

```
#undef sqrt
#define sqrt(x) __BUILTIN_GENERIC_sqrt(x)
```

Generic macros are designed for a useful level of consistency with C++ overloaded math functions.

The great majority of existing C programs are expected to be unaffected when the **<tgmath.h>** header is included instead of the **<math.h>** or **<complex.h>** headers. Generic macros are similar to the ISO/IEC 9899:1999 standard library masking macros, though the semantic types of return values differ.

The ability to overload on integer as well as floating types would have been useful for some functions; for example, *copysign*( ). Overloading with different numbers of arguments would have allowed reusing names; for example, *remainder*( ) for *remquo*( ). However, these facilities would have complicated the specification; and their natural consistent use, such as for a floating *abs*( ) or a two-argument *atan*( ), would have introduced further inconsistencies with the ISO/IEC 9899:1999 standard for insufficient benefit.

The ISO C standard in no way limits the implementation's options for efficiency, including inlining library functions.

FUTURE DIRECTIONS

None.

SEE ALSO

<math.h>, **<complex.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *cabs*( ), *fabs*( ), *modf*( )

CHANGE HISTORY

First released in Issue 6. Included for alignment with the ISO/IEC 9899:1999 standard.

NAME

time.h — time types

SYNOPSIS

```
#include <time.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<time.h>** header shall declare the structure **tm**, which shall include at least the following members:

`int`	`tm_sec`	Seconds [0,60].
`int`	`tm_min`	Minutes [0,59].
`int`	`tm_hour`	Hour [0,23].
`int`	`tm_mday`	Day of month [1,31].
`int`	`tm_mon`	Month of year [0,11].
`int`	`tm_year`	Years since 1900.
`int`	`tm_wday`	Day of week [0,6] (Sunday =0).
`int`	`tm_yday`	Day of year [0,365].
`int`	`tm_isdst`	Daylight savings flag.

The value of *tm_isdst* shall be positive if Daylight Saving Time is in effect, 0 if Daylight Saving Time is not in effect, and negative if the information is not available.

The **<time.h>** header shall define the following symbolic names:

NULL	Null pointer constant.
CLOCKS_PER_SEC	A number used to convert the value returned by the *clock*( ) function into seconds.
TMR|CPT CLOCK_PROCESS_CPUTIME_ID	The identifier of the CPU-time clock associated with the process making a *clock*( ) or *timer**( ) function call.
TMR|TCT CLOCK_THREAD_CPUTIME_ID	The identifier of the CPU-time clock associated with the thread making a *clock*( ) or *timer**( ) function call.

TMR The **<time.h>** header shall declare the structure **timespec**, which has at least the following members:

`time_t`	`tv_sec`	Seconds.
`long`	`tv_nsec`	Nanoseconds.

The **<time.h>** header shall also declare the **itimerspec** structure, which has at least the following members:

`struct timespec`	`it_interval`	Timer period.
`struct timespec`	`it_value`	Timer expiration.

The following manifest constants shall be defined:

CLOCK_REALTIME	The identifier of the system-wide realtime clock.
TIMER_ABSTIME	Flag indicating time is absolute with respect to the clock associated with a timer.

MON CLOCK_MONOTONIC
The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via *clock_settime*() and which cannot have backward clock jumps. The maximum possible clock jump shall be implementation-defined.

TMR The **clock_t**, **size_t**, **time_t**, **clockid_t**, and **timer_t** types shall be defined as described in **<sys/types.h>**.

XSI Although the value of CLOCKS_PER_SEC is required to be 1 million on all XSI-conformant systems, it may be variable on other systems, and it should not be assumed that CLOCKS_PER_SEC is a compile-time constant.

XSI The **<time.h>** header shall provide a declaration for *getdate_err*.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
char      *asctime(const struct tm *);
char      *asctime_r(const struct tm *restrict, char *restrict);
clock_t    clock(void);
int        clock_getcpuclockid(pid_t, clockid_t *);
int        clock_getres(clockid_t, struct timespec *);
int        clock_gettime(clockid_t, struct timespec *);
int        clock_nanosleep(clockid_t, int, const struct timespec *,
               struct timespec *);
int        clock_settime(clockid_t, const struct timespec *);
char      *ctime(const time_t *);
char      *ctime_r(const time_t *, char *);
double     difftime(time_t, time_t);
struct tm *getdate(const char *);
struct tm *gmtime(const time_t *);
struct tm *gmtime_r(const time_t *restrict, struct tm *restrict);
struct tm *localtime(const time_t *);
struct tm *localtime_r(const time_t *restrict, struct tm *restrict);
time_t     mktime(struct tm *);
int        nanosleep(const struct timespec *, struct timespec *);
size_t     strftime(char *restrict, size_t, const char *restrict,
           const struct tm *restrict);
char      *strptime(const char *restrict, const char *restrict,
               struct tm *restrict);
time_t     time(time_t *);
int        timer_create(clockid_t, struct sigevent *restrict,
               timer_t *restrict);
int        timer_delete(timer_t);
int        timer_gettime(timer_t, struct itimerspec *);
int        timer_getoverrun(timer_t);
int        timer_settime(timer_t, int, const struct itimerspec *restrict,
               struct itimerspec *restrict);
void       tzset(void);
```

The following shall be declared as variables:

```
extern int    daylight;
extern long   timezone;
```

CX
```
extern char   *tzname[];
```

CX Inclusion of the **<time.h>** header may make visible all symbols from the **<signal.h>** header.

APPLICATION USAGE

The range [0,60] for *tm_sec* allows for the occasional leap second.

tm_year is a signed value; therefore, years before 1900 may be represented.

To obtain the number of clock ticks per second returned by the *times*( ) function, applications should call *sysconf*(_SC_CLK_TCK).

RATIONALE

The range [0,60] seconds allows for positive or negative leap seconds. The formal definition of UTC does not permit double leap seconds, so all mention of double leap seconds has been removed, and the range shortened from the former [0,61] seconds seen in previous versions of POSIX.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *asctime*( ), *clock*( ), *clock_getcpuclockid*( ), *clock_getres*( ), *clock_nanosleep*( ), *ctime*( ), *difftime*( ), *getdate*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *nanosleep*( ), *strftime*( ), *strptime*( ), *sysconf*( ), *time*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *tzname*, *tzset*( ), *utime*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The Open Group Corrigendum U035/6 is applied. In the DESCRIPTION, the types **clockid_t** and **timer_t** have been described.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The POSIX timer-related functions are now marked as part of the Timers option.

The symbolic name CLK_TCK is removed. Application usage is added describing how its equivalent functionality can be obtained using *sysconf*( ).

The *clock_getcpuclockid*( ) function and manifest constants CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID are added for alignment with IEEE Std 1003.1d-1999.

The manifest constant CLOCK_MONOTONIC and the *clock_nanosleep*( ) function are added for alignment with IEEE Std 1003.1j-2000.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The range for seconds is changed from [0,61] to [0,60].
- The **restrict** keyword is added to the prototypes for *asctime_r*( ), *gmtime_r*( ), *localtime_r*( ), *strftime*( ), *strptime*( ), *timer_create*( ), and *timer_settime*( ).

IEEE PASC Interpretation 1003.1 #84 is applied adding the statement that symbols from the **<signal.h>** header may be made visible when the **<time.h>** header is included.

Extensions beyond the ISO C standard are now marked.

NAME

trace.h — tracing

SYNOPSIS

TRC

```
#include <trace.h>
```

DESCRIPTION

The **<trace.h>** header shall define the **posix_trace_event_info** structure that includes at least the following members:

```
trace_event_id_t  posix_event_id
pid_t             posix_pid
void             *posix_prog_address
int               posix_truncation_status
struct timespec   posix_timestamp
```

THR

```
pthread_t         posix_thread_id
```

The **<trace.h>** header shall define the **posix_trace_status_info** structure that includes at least the following members:

```
int     posix_stream_status
int     posix_stream_full_status
int     posix_stream_overrun_status
```

TRL

```
int     posix_stream_flush_status
int     posix_stream_flush_error
int     posix_log_overrun_status
int     posix_log_full_status
```

The **<trace.h>** header shall define the following symbols:

POSIX_TRACE_RUNNING
POSIX_TRACE_SUSPENDED
POSIX_TRACE_FULL
POSIX_TRACE_NOT_FULL
POSIX_TRACE_NO_OVERRUN
POSIX_TRACE_OVERRUN
TRL POSIX_TRACE_FLUSHING
POSIX_TRACE_NOT_FLUSHING
POSIX_TRACE_NOT_TRUNCATED
POSIX_TRACE_TRUNCATED_READ
POSIX_TRACE_TRUNCATED_RECORD
TRL POSIX_TRACE_FLUSH
POSIX_TRACE_LOOP
POSIX_TRACE_UNTIL_FULL
TRI POSIX_TRACE_CLOSE_FOR_CHILD
POSIX_TRACE_INHERITED
TRL POSIX_TRACE_APPEND
POSIX_TRACE_LOOP
POSIX_TRACE_UNTIL_FULL
TEF POSIX_TRACE_FILTER
TRL POSIX_TRACE_FLUSH_START
POSIX_TRACE_FLUSH_STOP
POSIX_TRACE_OVERFLOW

POSIX_TRACE_RESUME
POSIX_TRACE_START
POSIX_TRACE_STOP
POSIX_TRACE_UNNAMED_USER_EVENT

The following types shall be defined as described in **<sys/types.h>**:

trace_attr_t
trace_id_t
trace_event_id_t
TEF **trace_event_set_t**

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
    int  posix_trace_attr_destroy(trace_attr_t *);
    int  posix_trace_attr_getclockres(const trace_attr_t *,
             struct timespec *);
    int  posix_trace_attr_getcreatetime(const trace_attr_t *,
             struct timespec *);
    int  posix_trace_attr_getgenversion(const trace_attr_t *, char *);
TRI int  posix_trace_attr_getinherited(const trace_attr_t *restrict,
             int *restrict);
TRL int  posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict,
             int *restrict);
    int  posix_trace_attr_getlogsize(const trace_attr_t *restrict,
             size_t *restrict);
    int  posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict,
             size_t *restrict);
    int  posix_trace_attr_getmaxsystemeventsize(const trace_attr_t *restrict,
             size_t *restrict);
    int  posix_trace_attr_getmaxusereventsize(const trace_attr_t *restrict,
             size_t, size_t *restrict);
    int  posix_trace_attr_getname(const trace_attr_t *, char *);
    int  posix_trace_attr_getstreamfullpolicy(const trace_attr_t *restrict,
             int *restrict);
    int  posix_trace_attr_getstreamsize(const trace_attr_t *restrict,
             size_t *restrict);
    int  posix_trace_attr_init(trace_attr_t *);
TRI int  posix_trace_attr_setinherited(trace_attr_t *, int);
TRL int  posix_trace_attr_setlogfullpolicy(trace_attr_t *, int);
    int  posix_trace_attr_setlogsize(trace_attr_t *, size_t);
    int  posix_trace_attr_setmaxdatasize(trace_attr_t *, size_t);
    int  posix_trace_attr_setname(trace_attr_t *, const char *);
    int  posix_trace_attr_setstreamsize(trace_attr_t *, size_t);
    int  posix_trace_attr_setstreamfullpolicy(trace_attr_t *, int);
    int  posix_trace_clear(trace_id_t);
TRL int  posix_trace_close(trace_id_t);
    int  posix_trace_create(pid_t, const trace_attr_t *restrict,
             trace_id_t *restrict);
TRL int  posix_trace_create_withlog(pid_t, const trace_attr_t *restrict,
             int, trace_id_t *restrict);
    void posix_trace_event(trace_event_id_t, const void *restrict, size_t);
```

```
         int  posix_trace_eventid_equal(trace_id_t, trace_event_id_t,
                  trace_event_id_t);
         int  posix_trace_eventid_get_name(trace_id_t, trace_event_id_t, char *);
         int  posix_trace_eventid_open(const char *restrict,
                  trace_event_id_t *restrict);
         int  posix_trace_eventtypelist_getnext_id(trace_id_t,
                  trace_event_id_t *restrict, int *restrict);
         int  posix_trace_eventtypelist_rewind(trace_id_t);
TEF      int  posix_trace_eventset_add(trace_event_id_t, trace_event_set_t *);
         int  posix_trace_eventset_del(trace_event_id_t, trace_event_set_t *);
         int  posix_trace_eventset_empty(trace_event_set_t *);
         int  posix_trace_eventset_fill(trace_event_set_t *, int);
         int  posix_trace_eventset_ismember(trace_event_id_t,
                  const trace_event_set_t *restrict, int *restrict);
         int  posix_trace_flush(trace_id_t);
         int  posix_trace_get_attr(trace_id_t, trace_attr_t *);
TEF      int  posix_trace_get_filter(trace_id_t, trace_event_set_t *);
         int  posix_trace_get_status(trace_id_t,
                  struct posix_trace_status_info *);
         int  posix_trace_getnext_event(trace_id_t,
                  struct posix_trace_event_info *restrict , void *restrict,
                  size_t, size_t *restrict, int *restrict);
TRL      int  posix_trace_open(int, trace_id_t *);
         int  posix_trace_rewind(trace_id_t);
TEF      int  posix_trace_set_filter(trace_id_t, const trace_event_set_t *, int);
         int  posix_trace_shutdown(trace_id_t);
         int  posix_trace_start(trace_id_t);
         int  posix_trace_stop(trace_id_t);
TMO      int  posix_trace_timedgetnext_event(trace_id_t,
                  struct posix_trace_event_info *restrict, void *restrict,
                  size_t, size_t *restrict, int *restrict,
                  const struct timespec *restrict);
TEF      int  posix_trace_trid_eventid_open(trace_id_t, const char *restrict,
                  trace_event_id_t *restrict);
         int  posix_trace_trygetnext_event(trace_id_t,
                  struct posix_trace_event_info *restrict, void *restrict, size_t,
                  size_t *restrict, int *restrict);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.11, Tracing, the System Interfaces volume of IEEE Std 1003.1-200x, *posix_trace_attr_destroy*( ), *posix_trace_attr_getclockres*( ), *posix_trace_attr_getcreatetime*( ), *posix_trace_attr_getgenversion*( ), *posix_trace_attr_getinherited*( ), *posix_trace_attr_getlogfullpolicy*( ), *posix_trace_attr_getlogsize*( ), *posix_trace_attr_getmaxdatasize*( ), *posix_trace_attr_getmaxsystemeventsize*( ), *posix_trace_attr_getmaxusereventsize*( ), *posix_trace_attr_getname*( ), *posix_trace_attr_getstreamfullpolicy*( ), *posix_trace_attr_getstreamsize*( ), *posix_trace_attr_init*( ), *posix_trace_attr_setinherited*( ), *posix_trace_attr_setlogfullpolicy*( ), *posix_trace_attr_setlogsize*( ), *posix_trace_attr_setmaxdatasize*( ), *posix_trace_attr_setname*( ), *posix_trace_attr_setstreamsize*( ), *posix_trace_attr_setstreamfullpolicy*( ), *posix_trace_clear*( ), *posix_trace_close*( ), *posix_trace_create*( ), *posix_trace_create_withlog*( ), *posix_trace_event*( ), *posix_trace_eventid_equal*( ), *posix_trace_eventid_get_name*( ), *posix_trace_eventid_open*( ), *posix_trace_eventtypelist_getnext_id*( ), *posix_trace_eventtypelist_rewind*( ), *posix_trace_eventset_add*( ), *posix_trace_eventset_del*( ), *posix_trace_eventset_empty*( ), *posix_trace_eventset_fill*( ), *posix_trace_eventset_ismember*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), *posix_trace_get_filter*( ), *posix_trace_get_status*( ), *posix_trace_getnext_event*( ), *posix_trace_open*( ), *posix_trace_rewind*( ), *posix_trace_set_filter*( ), *posix_trace_shutdown*( ), *posix_trace_start*( ), *posix_trace_stop*( ), *posix_trace_timedgetnext_event*( ), *posix_trace_trid_eventid_open*( ), *posix_trace_trygetnext_event*( )

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

NAME

ucontext.h — user context

SYNOPSIS

XSI
```
#include <ucontext.h>
```

DESCRIPTION

The **<ucontext.h>** header shall define the **mcontext_t** type through **typedef**.

The **<ucontext.h>** header shall define the **ucontext_t** type as a structure that shall include at least the following members:

`ucontext_t`	`*uc_link`	Pointer to the context that is resumed when this context returns.
`sigset_t`	`uc_sigmask`	The set of signals that are blocked when this context is active.
`stack_t`	`uc_stack`	The stack used by this context.
`mcontext_t`	`uc_mcontext`	A machine-specific representation of the saved context.

The types **sigset_t** and **stack_t** shall be defined as in **<signal.h>**.

The following shall be declared as functions and may also be defined as macros, Function prototypes shall be provided.

```
int  getcontext(ucontext_t *);
int  setcontext(const ucontext_t *);
void makecontext(ucontext_t *, void (*)(void), int, ...);
int  swapcontext(ucontext_t *restrict, const ucontext_t *restrict);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<signal.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *getcontext*( ), *makecontext*( ), *sigaction*( ), *sigprocmask*( ), *sigaltstack*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

ulimit.h — ulimit commands

SYNOPSIS

XSI

```
#include <ulimit.h>
```

DESCRIPTION

The **<ulimit.h>** header shall define the symbolic constants used by the *ulimit*( ) function.

Symbolic constants:

UL_GETFSIZE	Get maximum file size.
UL_SETFSIZE	Set maximum file size.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided. |

```
long ulimit(int, ...);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *ulimit*( )

CHANGE HISTORY

First released in Issue 3.

NAME

unistd.h — standard symbolic constants and types

SYNOPSIS

```
#include <unistd.h>
```

DESCRIPTION

The **<unistd.h>** header defines miscellaneous symbolic constants and types, and declares miscellaneous functions. The actual value of the constants are unspecified except as shown. The contents of this header are shown below.

Version Test Macros

The following symbolic constants shall be defined:

_POSIX_VERSION
Integer value indicating version of IEEE Std 1003.1 (C-language binding) to which the implementation conforms. For implementations conforming to IEEE Std 1003.1-200x, the value shall be 200xxxL.

_POSIX2_VERSION
Integer value indicating version of the Shell and Utilities volume of IEEE Std 1003.1 to which the implementation conforms. For implementations conforming to IEEE Std 1003.1-200x, the value shall be 200xxxL.

The following symbolic constant shall be defined only if the implementation supports the XSI option; see Section 2.1.4 (on page 19).

XSI _XOPEN_VERSION
Integer value indicating version of the X/Open Portability Guide to which the implementation conforms. The value shall be 600.

Constants for Options and Option Groups

The following symbolic constants, if defined in **<unistd.h>**, shall have a value of –1, 0, or greater, unless otherwise specified below. If these are undefined, the *fpathconf*( ), *pathconf*( ), or *sysconf*( ) functions can be used to determine whether the option is provided for a particular invocation of the application.

If a symbolic constant is defined with the value –1, the option is not supported. Headers, data types, and function interfaces required only for the option need not be supplied. An application that attempts to use anything associated only with the option is considered to be requiring an extension.

If a symbolic constant is defined with a value greater than zero, the option shall always be supported when the application is executed. All headers, data types, and functions shall be present and shall operate as specified.

If a symbolic constant is defined with the value zero, all headers, data types, and functions shall be present. The application can check at runtime to see whether the option is supported by calling *fpathconf*( ), *pathconf*( ), or *sysconf*( ) with the indicated *name* parameter.

Unless explicitly specified otherwise, the behavior of functions associated with an unsupported option is unspecified, and an application that uses such functions without first checking *fpathconf*( ), *pathconf*( ), or *sysconf*( ) is considered to be requiring an extension.

For conformance requirements, refer to Chapter 2 (on page 15).

ADV _POSIX_ADVISORY_INFO

The implementation supports the Advisory Information option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

AIO _POSIX_ASYNCHRONOUS_IO

The implementation supports the Asynchronous Input and Output option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BAR _POSIX_BARRIERS

The implementation supports the Barriers option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX_CHOWN_RESTRICTED

The use of *chown*( ) and *fchown*( ) is restricted to a process with appropriate privileges, and to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs. This symbol shall always be set to a value other than –1.

CS _POSIX_CLOCK_SELECTION

The implementation supports the Clock Selection option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

CPT _POSIX_CPUTIME

The implementation supports the Process CPU-Time Clocks option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

FSC _POSIX_FSYNC

The implementation supports the File Synchronization option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX_JOB_CONTROL

The implementation supports job control. This symbol shall always be set to a value greater than zero.

MF _POSIX_MAPPED_FILES

The implementation supports the Memory Mapped Files option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

ML _POSIX_MEMLOCK

The implementation supports the Process Memory Locking option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

MLR _POSIX_MEMLOCK_RANGE

The implementation supports the Range Memory Locking option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

MPR _POSIX_MEMORY_PROTECTION

The implementation supports the Memory Protection option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

MSG _POSIX_MESSAGE_PASSING

The implementation supports the Message Passing option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

MON _POSIX_MONOTONIC_CLOCK

The implementation supports the Monotonic Clock option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX_NO_TRUNC

Pathname components longer than {NAME_MAX} generate an error. This symbol shall always be set to a value other than –1.

PIO _POSIX_PRIORITIZED_IO

The implementation supports the Prioritized Input and Output option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

PS _POSIX_PRIORITY_SCHEDULING

The implementation supports the Process Scheduling option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

RS _POSIX_RAW_SOCKETS

The implementation supports the Raw Sockets option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

THR _POSIX_READER_WRITER_LOCKS

The implementation supports the Read-Write Locks option. This is always set to a value greater than zero if the Threads option is supported. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

RTS _POSIX_REALTIME_SIGNALS

The implementation supports the Realtime Signals Extension option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX_REGEXP

The implementation supports the Regular Expression Handling option. This symbol shall always be set to a value greater than zero.

_POSIX_SAVED_IDS

Each process has a saved set-user-ID and a saved set-group-ID. This symbol shall always be set to a value greater than zero.

SEM _POSIX_SEMAPHORES

The implementation supports the Semaphores option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

SHM _POSIX_SHARED_MEMORY_OBJECTS

The implementation supports the Shared Memory Objects option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX_SHELL

The implementation supports the POSIX shell. This symbol shall always be set to a value greater than zero.

SPN _POSIX_SPAWN

The implementation supports the Spawn option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

SPI _POSIX_SPIN_LOCKS

The implementation supports the Spin Locks option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

SS _POSIX_SPORADIC_SERVER

The implementation supports the Process Sporadic Server option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

SIO _POSIX_SYNCHRONIZED_IO

The implementation supports the Synchronized Input and Output option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TSA _POSIX_THREAD_ATTR_STACKADDR

The implementation supports the Thread Stack Address Attribute option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TSS _POSIX_THREAD_ATTR_STACKSIZE

The implementation supports the Thread Stack Address Size option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TCT _POSIX_THREAD_CPUTIME

The implementation supports the Thread CPU-Time Clocks option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TPI _POSIX_THREAD_PRIO_INHERIT

The implementation supports the Thread Priority Inheritance option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TPP _POSIX_THREAD_PRIO_PROTECT

The implementation supports the Thread Priority Protection option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TPS _POSIX_THREAD_PRIORITY_SCHEDULING

The implementation supports the Thread Execution Scheduling option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TSH _POSIX_THREAD_PROCESS_SHARED

The implementation supports the Thread Process-Shared Synchronization option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TSF _POSIX_THREAD_SAFE_FUNCTIONS

The implementation supports the Thread-Safe Functions option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TSP _POSIX_THREAD_SPORADIC_SERVER

The implementation supports the Thread Sporadic Server option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

THR _POSIX_THREADS

The implementation supports the Threads option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TMO _POSIX_TIMEOUTS

The implementation supports the Timeouts option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TMR _POSIX_TIMERS

The implementation supports the Timers option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TRC _POSIX_TRACE

The implementation supports the Trace option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TEF _POSIX_TRACE_EVENT_FILTER

The implementation supports the Trace Event Filter option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TRI _POSIX_TRACE_INHERIT

The implementation supports the Trace Inherit option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TRL _POSIX_TRACE_LOG

The implementation supports the Trace Log option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

TYM _POSIX_TYPED_MEMORY_OBJECTS

The implementation supports the Typed Memory Objects option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX_VDISABLE

This symbol shall be defined to be the value of a character that shall disable terminal special character handling as described in **<termios.h>**. This symbol shall always be set to a value other than –1.

_POSIX2_C_BIND

The implementation supports the C-Language Binding option. This symbol shall always have the value 200xxxL.

CD _POSIX2_C_DEV

The implementation supports the C-Language Development Utilities option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX2_CHAR_TERM

The implementation supports at least one terminal type.

FD _POSIX2_FORT_DEV

The implementation supports the FORTRAN Development Utilities option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

FR _POSIX2_FORT_RUN

The implementation supports the FORTRAN Runtime Utilities option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_POSIX2_LOCALEDEF

The implementation supports the creation of locales by the *localedef* utility. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BE _POSIX2_PBS

The implementation supports the Batch Environment Services and Utilities option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BE _POSIX2_PBS_ACCOUNTING

The implementation supports the Batch Accounting option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BE _POSIX2_PBS_CHECKPOINT

The implementation supports the Batch Checkpoint/Restart option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BE _POSIX2_PBS_LOCATE

The implementation supports the Locate Batch Job Request option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BE _POSIX2_PBS_MESSAGE

The implementation supports the Batch Job Message Request option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

BE _POSIX2_PBS_TRACK

The implementation supports the Track Batch Job Request option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

SD _POSIX2_SW_DEV

The implementation supports the Software Development Utilities option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

UP _POSIX2_UPE

The implementation supports the User Portability Utilities option. If this symbol has a value other than –1 or 0, it shall have the value 200xxxL.

_V6_ILP32_OFF32

The implementation provides a C-language compilation environment with 32-bit **int**, **long**, **pointer**, and **off_t** types.

_V6_ILP32_OFFBIG

The implementation provides a C-language compilation environment with 32-bit **int**, **long**, and **pointer** types and an **off_t** type using at least 64 bits.

_V6_LP64_OFF64

The implementation provides a C-language compilation environment with 32-bit **int** and 64-bit **long**, **pointer**, and **off_t** types.

_V6_LPBIG_OFFBIG

The implementation provides a C-language compilation environment with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

XSI _XBS5_ILP32_OFF32 (**LEGACY**)

The implementation provides a C-language compilation environment with 32-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _XBS5_ILP32_OFFBIG (**LEGACY**)

The implementation provides a C-language compilation environment with 32-bit **int**, **long**, and **pointer** types and an **off_t** type using at least 64 bits.

XSI _XBS5_LP64_OFF64 (**LEGACY**)

The implementation provides a C-language compilation environment with 32-bit **int** and 64-bit **long**, **pointer**, and **off_t** types.

XSI _XBS5_LPBIG_OFFBIG (**LEGACY**)

The implementation provides a C-language compilation environment with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

XSI _XOPEN_CRYPT

The implementation supports the X/Open Encryption Option Group.

_XOPEN_ENH_I18N

The implementation supports the Issue 4, Version 2 Enhanced Internationalization Option Group. This symbol shall always be set to a value other than –1.

_XOPEN_LEGACY

The implementation supports the Legacy Option Group.

_XOPEN_REALTIME

The implementation supports the X/Open Realtime Option Group.

_XOPEN_REALTIME_THREADS

The implementation supports the X/Open Realtime Threads Option Group.

_XOPEN_SHM

The implementation supports the Issue 4, Version 2 Shared Memory Option Group. This symbol shall always be set to a value other than –1.

_XOPEN_STREAMS
The implementation supports the XSI STREAMS Option Group.

XSI _XOPEN_UNIX
The implementation supports the XSI extension.

Execution-Time Symbolic Constants

If any of the following constants are not defined in the **<unistd.h>** header, the value shall vary depending on the file to which it is applied.

If any of the following constants are defined to have value −1 in the **<unistd.h>** header, the implementation shall not provide the option on any file; if any are defined to have a value other than −1 in the **<unistd.h>** header, the implementation shall provide the option on all applicable files.

All of the following constants, whether defined in **<unistd.h>** or not, may be queried with respect to a specific file using the *pathconf*( ) or *fpathconf*( ) functions:

_POSIX_ASYNC_IO
Asynchronous input or output operations may be performed for the associated file.

_POSIX_PRIO_IO
Prioritized input or output operations may be performed for the associated file.

_POSIX_SYNC_IO
Synchronized input or output operations may be performed for the associated file.

Constants for Functions

The following symbolic constant shall be defined:

NULL	Null pointer

The following symbolic constants shall be defined for the *access*( ) function:

F_OK	Test for existence of file.
R_OK	Test for read permission.
W_OK	Test for write permission.
X_OK	Test for execute (search) permission.

The constants F_OK, R_OK, W_OK, and X_OK and the expressions *R_OK* | *W_OK*, *R_OK* | *X_OK*, and *R_OK* | *W_OK* | *X_OK* shall all have distinct values.

The following symbolic constants shall be defined for the *confstr*( ) function:

_CS_POSIX_PATH
This is the value for the *PATH* environment variable that finds all standard utilities.

_CS_POSIX_V6_ILP32_OFF32_CFLAGS
If *sysconf*(_SC_V6_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_ILP32_OFF32_LDFLAGS
If *sysconf*(_SC_V6_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_ILP32_OFF32_LIBS

If *sysconf*(_SC_V6_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_ILP32_OFF32_LINTFLAGS

If *sysconf*(_SC_V6_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check application source using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS

If *sysconf*(_SC_V6_ILP32_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS

If *sysconf*(_SC_V6_ILP32_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

_CS_POSIX_V6_ILP32_OFFBIG_LIBS

If *sysconf*(_SC_V6_ILP32_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

_CS_POSIX_V6_ILP32_OFFBIG_LINTFLAGS

If *sysconf*(_SC_V6_ILP32_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

_CS_POSIX_V6_LP64_OFF64_CFLAGS

If *sysconf*(_SC_V6_LP64_OFF64) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_LP64_OFF64_LDFLAGS

If *sysconf*(_SC_V6_LP64_OFF64) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_LP64_OFF64_LIBS

If *sysconf*(_SC_V6_LP64_OFF64) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_LP64_OFF64_LINTFLAGS

If *sysconf*(_SC_V6_LP64_OFF64) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check application source using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS

If *sysconf*(_SC_V6_LPBIG_OFFBIG) returns −1, the meaning of this value is unspecified.

Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS

If *sysconf*(_SC_V6_LPBIG_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

_CS_POSIX_V6_LPBIG_OFFBIG_LIBS

If *sysconf*(_SC_V6_LPBIG_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

_CS_POSIX_V6_LPBIG_OFFBIG_LINTFLAGS

If *sysconf*(_SC_V6_LPBIG_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check application source using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

_CS_V6_WIDTH_RESTRICTED_ENVS

This value is a <newline>-separated list of names of programming environments supported by the implementation in which the widths of the **blksize_t**, **cc_t**, **mode_t**, **nfds_t**, **pid_t**, **ptrdiff_t**, **size_t**, **speed_t**, **ssize_t**, **suseconds_t**, **tcflag_t**, **useconds_t**, **wchar_t**, and **wint_t** types are no greater than the width of type **long**.

XSI _CS_XBS5_ILP32_OFF32_CFLAGS (**LEGACY**)

If *sysconf*(_SC_XBS5_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_ILP32_OFF32_LDFLAGS (**LEGACY**)

If *sysconf*(_SC_XBS5_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_ILP32_OFF32_LIBS (**LEGACY**)

If *sysconf*(_SC_XBS5_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_ILP32_OFF32_LINTFLAGS (**LEGACY**)

If *sysconf*(_SC_XBS5_ILP32_OFF32) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check application source using a programming model with 32-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_ILP32_OFFBIG_CFLAGS (**LEGACY**)

If *sysconf*(_SC_XBS5_ILP32_OFFBIG) returns −1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

XSI _CS_XBS5_ILP32_OFFBIG_LDFLAGS (**LEGACY**)

If *sysconf*(_SC_XBS5_ILP32_OFFBIG) returns −1, the meaning of this value is unspecified.

Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

XSI _CS_XBS5_ILP32_OFFBIG_LIBS (**LEGACY**)
If *sysconf*(_SC_XBS5_ILP32_OFFBIG) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

XSI _CS_XBS5_ILP32_OFFBIG_LINTFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_ILP32_OFFBIG) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check an application using a programming model with 32-bit **int**, **long**, and **pointer** types, and an **off_t** type using at least 64 bits.

XSI _CS_XBS5_LP64_OFF64_CFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_LP64_OFF64) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_LP64_OFF64_LDFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_LP64_OFF64) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_LP64_OFF64_LIBS (**LEGACY**)
If *sysconf*(_SC_XBS5_LP64_OFF64) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_LP64_OFF64_LINTFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_LP64_OFF64) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the *lint* utility to check application source using a programming model with 64-bit **int**, **long**, **pointer**, and **off_t** types.

XSI _CS_XBS5_LPBIG_OFFBIG_CFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_LPBIG_OFFBIG) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the *cc* and *c99* utilities to build an application using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

XSI _CS_XBS5_LPBIG_OFFBIG_LDFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_LPBIG_OFFBIG) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the *cc* and *c99* utilities to build an application using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

XSI _CS_XBS5_LPBIG_OFFBIG_LIBS (**LEGACY**)
If *sysconf*(_SC_XBS5_LPBIG_OFFBIG) returns –1, the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the *cc* and *c99* utilities to build an application using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

XSI _CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (**LEGACY**)
If *sysconf*(_SC_XBS5_LPBIG_OFFBIG) returns –1, the meaning of this value is unspecified.

Otherwise, this value is the set of options to be given to the *lint* utility to check application source using a programming model with an **int** type using at least 32 bits and **long**, **pointer**, and **off_t** types using at least 64 bits.

The following symbolic constants shall be defined for the *lseek*( ) and *fcntl*( ) functions and shall have distinct values:

SEEK_CUR	Set file offset to current plus *offset*.
SEEK_END	Set file offset to EOF plus *offset*.
SEEK_SET	Set file offset to *offset*.

The following symbolic constants shall be defined as possible values for the *function* argument to the *lockf*( ) function:

F_LOCK	Lock a section for exclusive use.
F_TEST	Test section for locks by other processes.
F_TLOCK	Test and lock a section for exclusive use.
F_ULOCK	Unlock locked sections.

The following symbolic constants shall be defined for *pathconf*( ):

ADV _PC_ALLOC_SIZE_MIN
AIO _PC_ASYNC_IO
_PC_CHOWN_RESTRICTED
_PC_FILESIZEBITS
_PC_LINK_MAX
_PC_MAX_CANON
_PC_MAX_INPUT
_PC_NAME_MAX
_PC_NO_TRUNC
_PC_PATH_MAX
_PC_PIPE_BUF
_PC_PRIO_IO
ADV _PC_REC_INCR_XFER_SIZE
_PC_REC_MAX_XFER_SIZE
_PC_REC_MIN_XFER_SIZE
_PC_REC_XFER_ALIGN
_PC_SYNC_IO
_PC_VDISABLE

The following symbolic constants shall be defined for *sysconf*( ):

_SC_2_C_BIND
_SC_2_C_DEV
_SC_2_C_VERSION
_SC_2_CHAR_TIME
_SC_2_FORT_DEV
_SC_2_FORT_RUN
_SC_2_LOCALEDEF
_SC_2_PBS
_SC_2_PBS_ACCOUNTING
_SC_2_PBS_CHECKPOINT
_SC_2_PBS_LOCATE
_SC_2_PBS_MESSAGE

_SC_2_PBS_TRACK
_SC_2_SW_DEV
_SC_2_UPE
_SC_2_VERSION
_SC_ADVISORY_INFO
_SC_ARG_MAX
_SC_AIO_LISTIO_MAX
_SC_AIO_MAX
_SC_AIO_PRIO_DELTA_MAX
_SC_ASYNCHRONOUS_IO
XSI _SC_ATEXIT_MAX
BAR _SC_BARRIERS
_SC_BASE
_SC_BC_BASE_MAX
_SC_BC_DIM_MAX
_SC_BC_SCALE_MAX
_SC_BC_STRING_MAX
_SC_C_LANG_SUPPORT
_SC_C_LANG_SUPPORT_R
_SC_CHILD_MAX
_SC_CLK_TCK
CS _SC_CLOCK_SELECTION
_SC_COLL_WEIGHTS_MAX
_SC_CPUTIME
_SC_DELAYTIMER_MAX
_SC_DEVICE_IO
_SC_DEVICE_SPECIFIC
_SC_DEVICE_SPECIFIC_R
_SC_EXPR_NEST_MAX
_SC_FD_MGMT
_SC_FIFO
_SC_FILE_ATTRIBUTES
_SC_FILE_LOCKING
_SC_FILE_SYSTEM
_SC_FSYNC
_SC_GETGR_R_SIZE_MAX
_SC_GETPW_R_SIZE_MAX
_SC_HOST_NAME_MAX
XSI _SC_IOV_MAX
_SC_JOB_CONTROL
_SC_LINE_MAX
_SC_LOGIN_NAME_MAX
_SC_MAPPED_FILES
_SC_MEMLOCK
_SC_MEMLOCK_RANGE
_SC_MEMORY_PROTECTION
_SC_MESSAGE_PASSING
MON _SC_MONOTONIC_CLOCK
_SC_MQ_OPEN_MAX
_SC_MQ_PRIO_MAX
_SC_MULTIPLE_PROCESS
_SC_NETWORKING

_SC_NGROUPS_MAX
_SC_OPEN_MAX
XSI _SC_PAGE_SIZE
_SC_PAGESIZE
_SC_PIPE
_SC_PRIORITIZED_IO
_SC_PRIORITY_SCHEDULING
_SC_RE_DUP_MAX
THR _SC_READER_WRITER_LOCKS
_SC_REALTIME_SIGNALS
_SC_REGEXP
_SC_RTSIG_MAX
_SC_SAVED_IDS
_SC_SEMAPHORES
_SC_SEM_NSEMS_MAX
_SC_SEM_VALUE_MAX
_SC_SHARED_MEMORY_OBJECTS
_SC_SHELL
_SC_SIGNALS
_SC_SIGQUEUE_MAX
_SC_SINGLE_PROCESS
_SC_SPAWN
SPI _SC_SPIN_LOCKS
_SC_SPORADIC_SERVER
_SC_STREAM_MAX
_SC_SYNCHRONIZED_IO
_SC_SYSTEM_DATABASE
_SC_SYSTEM_DATABASE_R
_SC_THREAD_ATTR_STACKADDR
_SC_THREAD_ATTR_STACKSIZE
_SC_THREAD_CPUTIME
_SC_THREAD_DESTRUCTOR_ITERATIONS
_SC_THREAD_KEYS_MAX
_SC_THREAD_PRIO_INHERIT
_SC_THREAD_PRIO_PROTECT
_SC_THREAD_PRIORITY_SCHEDULING
_SC_THREAD_PROCESS_SHARED
_SC_THREAD_SAFE_FUNCTIONS
_SC_THREAD_SPORADIC_SERVER
_SC_THREAD_STACK_MIN
_SC_THREAD_THREADS_MAX
_SC_TIMEOUTS
_SC_THREADS
_SC_TIMER_MAX
_SC_TIMERS
TRC _SC_TRACE
TEF _SC_TRACE_EVENT_FILTER
TRI _SC_TRACE_INHERIT
TRL _SC_TRACE_LOG
_SC_TTY_NAME_MAX
TYM _SC_TYPED_MEMORY_OBJECTS
_SC_TZNAME_MAX

_SC_USER_GROUPS
_SC_USER_GROUPS_R
_SC_V6_ILP32_OFF32
_SC_V6_ILP32_OFFBIG
_SC_V6_LP64_OFF64
_SC_V6_LPBIG_OFFBIG
_SC_VERSION
XSI _SC_XBS5_ILP32_OFF32 (**LEGACY**)
_SC_XBS5_ILP32_OFFBIG (**LEGACY**)
_SC_XBS5_LP64_OFF64 (**LEGACY**)
_SC_XBS5_LPBIG_OFFBIG (**LEGACY**)
_SC_XOPEN_CRYPT
_SC_XOPEN_ENH_I18N
_SC_XOPEN_LEGACY
_SC_XOPEN_REALTIME
_SC_XOPEN_REALTIME_THREADS
_SC_XOPEN_SHM
_SC_XOPEN_STREAMS
_SC_XOPEN_UNIX
_SC_XOPEN_VERSION
_SC_XOPEN_XCU_VERSION

The two constants _SC_PAGESIZE and _SC_PAGE_SIZE may be defined to have the same value.

The following symbolic constants shall be defined for file streams:

STDERR_FILENO	File number of *stderr*; 2.
STDIN_FILENO	File number of *stdin*; 0.
STDOUT_FILENO	File number of *stdout*; 1.

Type Definitions

The **size_t**, **ssize_t**, **uid_t**, **gid_t**, **off_t**, and **pid_t** types shall be defined as described in **<sys/types.h>**.

The **useconds_t** type shall be defined as described in **<sys/types.h>**.

The **intptr_t** type shall be defined as described in **<inttypes.h>**.

Declarations

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int          access(const char *, int);
unsigned     alarm(unsigned);
int          chdir(const char *);
int          chown(const char *, uid_t, gid_t);
int          close(int);
size_t       confstr(int, char *, size_t);
char        *crypt(const char *, const char *);
char        *ctermid(char *);
int          dup(int);
```

```
    int           dup2(int, int);
XSI void          encrypt(char[64], int);
    int           execl(const char *, const char *, ...);
    int           execle(const char *, const char *, ...);
    int           execlp(const char *, const char *, ...);
    int           execv(const char *, char *const []);
    int           execve(const char *, char *const [], char *const []);
    int           execvp(const char *, char *const []);
    void         _exit(int);
    int           fchown(int, uid_t, gid_t);
XSI int           fchdir(int);
SIO int           fdatasync(int);
    pid_t         fork(void);
    long          fpathconf(int, int);
    int           fsync(int);
    int           ftruncate(int, off_t);
    char         *getcwd(char *, size_t);
    gid_t         getegid(void);
    uid_t         geteuid(void);
    gid_t         getgid(void);
    int           getgroups(int, gid_t []);
XSI long          gethostid(void);
    int           gethostname(char *, size_t);
    char         *getlogin(void);
    int           getlogin_r(char *, size_t);
    int           getopt(int, char * const [], const char *);
XSI pid_t         getpgid(pid_t);
    pid_t         getpgrp(void);
    pid_t         getpid(void);
    pid_t         getppid(void);
XSI pid_t         getsid(pid_t);
    uid_t         getuid(void);
XSI char         *getwd(char *); (LEGACY)
    int           isatty(int);
XSI int           lchown(const char *, uid_t, gid_t);
    int           link(const char *, const char *);
XSI int           lockf(int, int, off_t);
    off_t         lseek(int, off_t, int);
XSI int           nice(int);
    long          pathconf(const char *, int);
    int           pause(void);
    int           pipe(int [2]);
XSI ssize_t       pread(int, void *, size_t, off_t);
    ssize_t       pwrite(int, const void *, size_t, off_t);
    ssize_t       read(int, void *, size_t);
    ssize_t       readlink(const char *restrict, char *restrict, size_t);
    int           rmdir(const char *);
    int           setegid(gid_t);
    int           seteuid(uid_t);
    int           setgid(gid_t);
```

```
int          setpgid(pid_t, pid_t);
pid_t        setpgrp(void);
int          setregid(gid_t, gid_t);
int          setreuid(uid_t, uid_t);
pid_t        setsid(void);
int          setuid(uid_t);
unsigned     sleep(unsigned);
void         swab(const void *restrict, void *restrict, ssize_t);
int          symlink(const char *, const char *);
void         sync(void);
long         sysconf(int);
pid_t        tcgetpgrp(int);
int          tcsetpgrp(int, pid_t);
int          truncate(const char *, off_t);
char        *ttyname(int);
int          ttyname_r(int, char *, size_t);
useconds_t   ualarm(useconds_t, useconds_t);
int          unlink(const char *);
int          usleep(useconds_t);
pid_t        vfork(void);
ssize_t      write(int, const void *, size_t);
```

Implementations may also include the *pthread_atfork*( ) prototype as defined in **<pthread.h>** (on page 286).

The following external variables shall be declared:

```
extern char   *optarg;
extern int    optind, opterr, optopt;
```

APPLICATION USAGE

IEEE Std 1003.1-200x only describes the behavior of systems that claim conformance to it. However, application developers who want to write applications that adapt to other versions of IEEE Std 1003.1 (or to systems that do not conform to any POSIX standard) may find it useful to code them so as to conditionally compile different code depending on the value of _POSIX_VERSION, for example.

```
#if _POSIX_VERSION == 200xxxL
/* Use the newer function that copes with large files. */
off_t pos=ftello(fp);
#else
/* Either this is an old version of POSIX, or _POSIX_VERSION is
   not even defined, so use the traditional function. */
long pos=ftell(fp);
#endif
```

Earlier versions of IEEE Std 1003.1 and of the Single UNIX Specification can be identified by the following macros:

POSIX.1-1988 standard
    _POSIX_VERSION==198808L

POSIX.1-1990 standard
    _POSIX_VERSION==199009L

ISO POSIX-1: 1996 standard
    _POSIX_VERSION==199506L

Single UNIX Specification, Version 1
_XOPEN_UNIX and _XOPEN_VERSION= =4

Single UNIX Specification, Version 2
_XOPEN_UNIX and _XOPEN_VERSION= =500

IEEE Std 1003.1-200x does not make any attempt to define application binary interaction with the underlying operating system. However, application developers may find it useful to query _SC_VERSION at runtime via *sysconf*( ) to determine whether the current version of the operating system supports the necessary functionality as in the following program fragment:

```
if (sysconf(_SC_VERSION) < 200xxxL) {
    fprintf(stderr, "POSIX.1-200x system required, terminating \n");
    exit(1);
}
```

RATIONALE

As IEEE Std 1003.1-200x evolved, certain options became sufficiently standardized that it was concluded that simply requiring one of the option choices was simpler than retaining the option. However, for backwards compatibility, the option flags (with required constant values) are retained.

Version Test Macros

The standard developers considered altering the definition of _POSIX_VERSION and removing _SC_VERSION from the specification of *sysconf*( ) since the utility to an application was deemed by some to be minimal, and since the implementation of the functionality is potentially problematic. However, they recognized that support for existing application binaries is a concern to manufacturers, application developers, and the users of implementations conforming to IEEE Std 1003.1-200x.

While the example using _SC_VERSION in the APPLICATION USAGE section does not provide the greatest degree of imaginable utility to the application developer or user, it is arguably better than a core dump or some other equally obscure result. (It is also possible for implementations to encode and recognize application binaries compiled in various POSIX.1-conforming environments, and modify the semantics of the underlying system to conform to the expectations of the application.) For the reasons outlined in the preceding paragraphs and in the APPLICATION USAGE section, the standard developers elected to retain the _POSIX_VERSION and _SC_VERSION functionality.

Compile-Time Symbolic Constants for System-Wide Options

IEEE Std 1003.1-200x now includes support in certain areas for the newly adopted policy governing options and stubs.

This policy provides flexibility for implementations in how they support options. It also specifies how conforming applications can adapt to different implementations that support different sets of options. It allows the following:

1. If an implementation has no interest in supporting an option, it does not have to provide anything associated with that option beyond the announcement that it does not support it.

2. An implementation can support a partial or incompatible version of an option (as a non-standard extension) as long as it does not claim to support the option.

3. An application can determine whether the option is supported. A strictly conforming application must check this announcement mechanism before first using anything associated with the option.

There is an important implication of this policy. IEEE Std 1003.1-200x cannot dictate the behavior of interfaces associated with an option when the implementation does not claim to support the option. In particular, it cannot require that a function associated with an unsupported option will fail if it does not perform as specified. However, this policy does not prevent a standard from requiring certain functions to always be present, but that they shall always fail on some implementations. The *setpgid*( ) function in the POSIX.1-1990 standard, for example, is considered appropriate.

The POSIX standards include various options, and the C language binding support for an option implies that the implementation must supply data types and function interfaces. An application must be able to discover whether the implementation supports each option.

Any application must consider the following three cases for each option:

1. Option never supported.

   The implementation advertises at compile time that the option will never be supported. In this case, it is not necessary for the implementation to supply any of the data types or function interfaces that are provided only as part of the option. The implementation might provide data types and functions that are similar to those defined by IEEE Std 1003.1-200x, but there is no guarantee for any particular behavior.

2. Option always supported.

   The implementation advertises at compile time that the option will always be supported. In this case, all data types and function interfaces shall be available and shall operate as specified.

3. Option might or might not be supported.

   Some implementations might not provide a mechanism to specify support of options at compile time. In addition, the implementation might be unable or unwilling to specify support or non-support at compile time. In either case, any application that might use the option at runtime must be able to compile and execute. The implementation must provide, at compile time, all data types and function interfaces that are necessary to allow this. In this situation, there must be a mechanism that allows the application to query, at runtime, whether the option is supported. If the application attempts to use the option when it is not supported, the result is unspecified unless explicitly specified otherwise in IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

<inttypes.h>, **<limits.h>**, **<sys/socket.h>**, **<sys/types.h>**, **<termios.h>**, **<wctype.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *access*( ), *alarm*( ), *chdir*( ), *chown*( ), *close*( ), *crypt*( ), *ctermid*( ), *dup*( ), *encrypt*( ), *environ*, *exec*( ), *exit*( ), *fchdir*( ), *fchown*( ), *fcntl*( ), *fork*( ), *fpathconf*( ), *fsync*( ), *ftruncate*( ), *getcwd*( ), *getegid*( ), *geteuid*( ), *getgid*( ), *getgroups*( ), *gethostid*( ), *gethostname*( ), *getlogin*( ), *getpgid*( ), *getpgrp*( ), *getpid*( ), *getppid*( ), *getsid*( ), *getuid*( ), *isatty*( ), *lchown*( ), *link*( ), *lockf*( ), *lseek*( ), *nice*( ), *pathconf*( ), *pause*( ), *pipe*( ), *read*( ), *readlink*( ), *rmdir*( ), *setgid*( ), *setpgid*( ), *setpgrp*( ), *setregid*( ), *setreuid*( ), *setsid*( ), *setuid*( ), *sleep*( ), *swab*( ), *symlink*( ), *sync*( ), *sysconf*( ), *tcgetpgrp*( ), *tcsetpgrp*( ), *truncate*( ), *ttyname*( ), *ualarm*( ), *unlink*( ), *usleep*( ), *vfork*( ), *write*( )

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

The symbolic constants _XOPEN_REALTIME and _XOPEN_REALTIME_THREADS are added. _POSIX2_C_BIND, _XOPEN_ENH_I18N, and _XOPEN_SHM must now be set to a value other than −1 by a conforming implementation.

Large File System extensions are added.

The type of the argument to *sbrk*( ) is changed from **int** to **intptr_t**.

XBS constants are added to the list of constants for Options and Option Groups, to the list of constants for the *confstr*( ) function, and to the list of constants to the *sysconf*( ) function. These are all marked EX.

Issue 6

_POSIX2_C_VERSION is removed.

The Open Group Corrigendum U026/4 is applied, adding the prototype for *fdatasync*( ).

The Open Group Corrigendum U026/1 is applied, adding the symbols _SC_XOPEN_LEGACY, _SC_XOPEN_REALTIME, and _SC_XOPEN_REALTIME_THREADS.

The symbols _XOPEN_STREAMS and _SC_XOPEN_STREAMS are added to support the XSI STREAMS Option Group.

Text in the DESCRIPTION relating to conformance requirements is moved elsewhere in IEEE Std 1003.1-200x. |

The legacy symbol _SC_PASS_MAX is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The _CS_POSIX_* and _CS_XBS5_* constants are added for the *confstr*( ) function. |
- The _SC_XBS5_* constants are added for the *sysconf*( ) function.
- The symbolic constants F_ULOCK, F_LOCK, F_TLOCK, and F_TEST are added.
- The **uid_t**, **gid_t**, **off_t**, **pid_t**, and **useconds_t** types are mandated.

The *gethostname*( ) prototype is added for sockets.

New section added for System Wide Options.

Function prototypes for *setegid*( ) and *seteuid*( ) are added.

Option symbolic constants are added for _POSIX_ADVISORY_INFO, _POSIX_CPUTIME, | _POSIX_SPAWN, _POSIX_SPORADIC_SERVER, _POSIX_THREAD_CPUTIME, _POSIX_THREAD_SPORADIC_SERVER, and _POSIX_TIMEOUTS, and *pathconf*( ) variables are added for _PC_ALLOC_SIZE_MIN, _PC_REC_INCR_XFER_SIZE, _PC_REC_MAX_XFER_SIZE, _PC_REC_MIN_XFER_SIZE, and _PC_REC_XFER_ALIGN for alignment with IEEE Std 1003.1d-1999. |

The following are added for alignment with IEEE Std 1003.1j-2000:

- Option symbolic constants _POSIX_BARRIERS, _POSIX_CLOCK_SELECTION, _POSIX_MONOTONIC_CLOCK, _POSIX_READER_WRITER_LOCKS, _POSIX_SPIN_LOCKS, and _POSIX_TYPED_MEMORY_OBJECTS

- *sysconf*( ) variables _SC_BARRIERS, _SC_CLOCK_SELECTION, _SC_MONOTONIC_CLOCK, _SC_READER_WRITER_LOCKS, _SC_SPIN_LOCKS, and _SC_TYPED_MEMORY_OBJECTS

The _SC_XBS5 macros associated with the ISO/IEC 9899:1990 standard are marked LEGACY, and new equivalent _SC_V6 macros associated with the ISO/IEC 9899:1999 standard are introduced.

The *getwd*( ) function is marked LEGACY.

The **restrict** keyword is added to the prototypes for *readlink*( ) and *swab*( ).

Constants for options are now harmonized, so when supported they take the year of approval of IEEE Std 1003.1-200x as the value.

The following are added for alignment with IEEE Std 1003.1q-2000:

- Optional symbolic constants _POSIX_TRACE, _POSIX_TRACE_EVENT_FILTER, _POSIX_TRACE_LOG, and _POSIX_TRACE_INHERIT
- The *sysconf*( ) symbolic constants _SC_TRACE, _SC_TRACE_EVENT_FILTER, _SC_TRACE_LOG, and _SC_TRACE_INHERIT.

The *brk*( ) and *sbrk*( ) legacy functions are removed.

The Open Group Base Resolution bwg2001-006 is applied, which reworks the XSI versioning information.

The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter for *gethostname*( ) from **socklen_t** to **size_t**.

<utime.h>

NAME

utime.h — access and modification times structure

SYNOPSIS

```
#include <utime.h>
```

DESCRIPTION

The **<utime.h>** header shall declare the structure **utimbuf**, which shall include the following members:

```
time_t    actime     Access time.
time_t    modtime    Modification time.
```

The times shall be measured in seconds since the Epoch.

The type **time_t** shall be defined as described in **<sys/types.h>**.

The following shall be declared as a function and may also be defined as a macro. A function prototype shall be provided. |

```
int utime(const char *, const struct utimbuf *);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/types.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *utime*( )

CHANGE HISTORY

First released in Issue 3.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **time_t** type is defined.

NAME

utmpx.h — user accounting database definitions

SYNOPSIS

XSI
```
#include <utmpx.h>
```

DESCRIPTION

The **<utmpx.h>** header shall define the **utmpx** structure that shall include at least the following members:

`char`	`ut_user[]`	User login name.
`char`	`ut_id[]`	Unspecified initialization process identifier.
`char`	`ut_line[]`	Device name.
`pid_t`	`ut_pid`	Process ID.
`short`	`ut_type`	Type of entry.
`struct timeval`	`ut_tv`	Time entry was made.

The **pid_t** type shall be defined through **typedef** as described in **<sys/types.h>**.

The **timeval** structure shall be defined as described in **<sys/time.h>**.

Inclusion of the **<utmpx.h>** header may also make visible all symbols from **<sys/time.h>**.

The following symbolic constants shall be defined as possible values for the *ut_type* member of the **utmpx** structure:

EMPTY	No valid user accounting information.
BOOT_TIME	Identifies time of system boot.
OLD_TIME	Identifies time when system clock changed.
NEW_TIME	Identifies time after system clock changed.
USER_PROCESS	Identifies a process.
INIT_PROCESS	Identifies a process spawned by the init process.
LOGIN_PROCESS	Identifies the session leader of a logged in user.
DEAD_PROCESS	Identifies a session leader who has exited.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
void          endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *);
struct utmpx *getutxline(const struct utmpx *);
struct utmpx *pututxline(const struct utmpx *);
void          setutxent(void);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<sys/time.h>, **<sys/types.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *endutxent*( )

CHANGE HISTORY

First released in Issue 4, Version 2.

NAME

wchar.h — wide-character handling

SYNOPSIS

```
#include <wchar.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<wchar.h>** header shall define the following types:

wchar_t	As described in **<stddef.h>**.
wint_t	An integer type capable of storing any valid value of **wchar_t** or WEOF.
XSI **wctype_t**	A scalar type of a data object that can hold values which represent locale-specific character classification.
mbstate_t	An object type other than an array type that can hold the conversion state information necessary to convert between sequences of (possibly multi-byte) characters and wide characters. XSI If a codeset is being used such that an **mbstate_t** needs to preserve more than 2 levels of reserved state, the results are unspecified.
XSI **FILE**	As described in **<stdio.h>**.
size_t	As described in **<stddef.h>**.
XSI **va_list**	As described in **<stdarg.h>**.

The implementation shall support one or more programming environments in which the width of **wint_t** is no greater than the width of type **long**. The names of these programming environments can be obtained using the *confstr*( ) function or the *getconf* utility.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
wint_t        btowc(int);
wint_t        fgetwc(FILE *);
wchar_t      *fgetws(wchar_t *restrict, int, FILE *restrict);
wint_t        fputwc(wchar_t, FILE *);
int           fputws(const wchar_t *restrict, FILE *restrict);
int           fwide(FILE *, int);
int           fwprintf(FILE *restrict, const wchar_t *restrict, ...);
int           fwscanf(FILE *restrict, const wchar_t *restrict, ...);
wint_t        getwc(FILE *);
wint_t        getwchar(void);
```

XSI

```
int           iswalnum(wint_t);
int           iswalpha(wint_t);
int           iswcntrl(wint_t);
int           iswctype(wint_t, wctype_t);
int           iswdigit(wint_t);
int           iswgraph(wint_t);
int           iswlower(wint_t);
int           iswprint(wint_t);
int           iswpunct(wint_t);
```

```
    int           iswspace(wint_t);
    int           iswupper(wint_t);
    int           iswxdigit(wint_t);
    size_t        mbrlen(const char *restrict, size_t, mbstate_t *restrict);
    size_t        mbrtowc(wchar_t *restrict, const char *restrict, size_t,
                      mbstate_t *restrict);
    int           mbsinit(const mbstate_t *);
    size_t        mbsrtowcs(wchar_t *restrict, const char **restrict, size_t,
                      mbstate_t *restrict);
    wint_t        putwc(wchar_t, FILE *);
    wint_t        putwchar(wchar_t);
    int           swprintf(wchar_t *restrict, size_t,
                      const wchar_t *restrict, ...);
    int           swscanf(const wchar_t *restrict,
                      const wchar_t *restrict, ...);
XSI wint_t        towlower(wint_t);
    wint_t        towupper(wint_t);
    wint_t        ungetwc(wint_t, FILE *);
    int           vfwprintf(FILE *restrict, const wchar_t *restrict, va_list);
    int           vfwscanf(FILE *restrict, const wchar_t *restrict, va_list);
    int           vwprintf(const wchar_t *restrict, va_list);
    int           vswprintf(wchar_t *restrict, size_t,
                      const wchar_t *restrict, va_list);
    int           vswscanf(const wchar_t *restrict, const wchar_t *restrict,
                      va_list);
    int           vwscanf(const wchar_t *restrict, va_list);
    size_t        wcrtomb(char *restrict, wchar_t, mbstate_t *restrict);
    wchar_t      *wcscat(wchar_t *restrict, const wchar_t *restrict);
    wchar_t      *wcschr(const wchar_t *, wchar_t);
    int           wcscmp(const wchar_t *, const wchar_t *);
    int           wcscoll(const wchar_t *, const wchar_t *);
    wchar_t      *wcscpy(wchar_t *restrict, const wchar_t *restrict);
    size_t        wcscspn(const wchar_t *, const wchar_t *);
    size_t        wcsftime(wchar_t *restrict, size_t,
                      const wchar_t *restrict, const struct tm *restrict);
    size_t        wcslen(const wchar_t *);
    wchar_t      *wcsncat(wchar_t *restrict, const wchar_t *restrict, size_t);
    int           wcsncmp(const wchar_t *, const wchar_t *, size_t);
    wchar_t      *wcsncpy(wchar_t *restrict, const wchar_t *restrict, size_t);
    wchar_t      *wcspbrk(const wchar_t *, const wchar_t *);
    wchar_t      *wcsrchr(const wchar_t *, wchar_t);
    size_t        wcsrtombs(char *restrict, const wchar_t **restrict,
                      size_t, mbstate_t *restrict);
    size_t        wcsspn(const wchar_t *, const wchar_t *);
    wchar_t      *wcsstr(const wchar_t *restrict, const wchar_t *restrict);
    double        wcstod(const wchar_t *restrict, wchar_t **restrict);
    float         wcstof(const wchar_t *restrict, wchar_t **restrict);
    wchar_t      *wcstok(wchar_t *restrict, const wchar_t *restrict,
                      wchar_t **restrict);
    long          wcstol(const wchar_t *restrict, wchar_t **restrict, int);
    long double   wcstold(const wchar_t *restrict, wchar_t **restrict);
    long long     wcstoll(const wchar_t *restrict, wchar_t **restrict, int);
```

```
unsigned long wcstoul(const wchar_t *restrict, wchar_t **restrict, int);
unsigned long long
              wcstoull(const wchar_t *restrict, wchar_t **restrict, int);
wchar_t      *wcswcs(const wchar_t *, const wchar_t *);
int           wcswidth(const wchar_t *, size_t);
size_t        wcsxfrm(wchar_t *restrict, const wchar_t *restrict, size_t);
int           wctob(wint_t);
wctype_t      wctype(const char *);
int           wcwidth(wchar_t);
wchar_t      *wmemchr(const wchar_t *, wchar_t, size_t);
int           wmemcmp(const wchar_t *, const wchar_t *, size_t);
wchar_t      *wmemcpy(wchar_t *restrict, const wchar_t *restrict, size_t);
wchar_t      *wmemmove(wchar_t *, const wchar_t *, size_t);
wchar_t      *wmemset(wchar_t *, wchar_t, size_t);
int           wprintf(const wchar_t *restrict, ...);
int           wscanf(const wchar_t *restrict, ...);
```

The **<wchar.h>** header shall define the following macros:

WCHAR_MAX	The maximum value representable by an object of type **wchar_t**.
WCHAR_MIN	The minimum value representable by an object of type **wchar_t**.
WEOF	Constant expression of type **wint_t** that is returned by several WP functions to indicate end-of-file.
NULL	As described in **<stddef.h>**.

The tag **tm** shall be declared as naming an incomplete structure type, the contents of which are described in the header **<time.h>**.

Inclusion of the **<wchar.h>** header may make visible all symbols from the headers **<ctype.h>**, **<stdio.h>**, **<stdarg.h>**, **<stdlib.h>**, **<string.h>**, **<stddef.h>**, and **<time.h>**.

APPLICATION USAGE

None.

RATIONALE

In the ISO C standard, the symbols referenced as XSI extensions are in **<wctype.h>**. Their presence here is thus an extension.

FUTURE DIRECTIONS

None.

SEE ALSO

<ctype.h>, **<stdarg.h>**, **<stddef.h>**, **<stdio.h>**, **<stdlib.h>**, **<string.h>**, **<time.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *btowc*( ), *confstr*( ), *fgetwc*( ), *fgetws*( ), *fputwc*( ), *fputws*( ), *fwide*( ), *fwprintf*( ), *fwscanf*( ), *getwc*( ), *getwchar*( ), *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *iswctype*( ), *mbsinit*( ), *mbrlen*( ), *mbrtowc*( ), *mbsrtowcs*( ), *putwc*( ), *putwchar*( ), *swprintf*( ), *swscanf*( ), *towlower*( ), *towupper*( ), *ungetwc*( ), *vfwprintf*( ), *vfwscanf*( ), *vswprintf*( ), *vswscanf*( ), *vwscanf*( ), *wcrtomb*( ), *wcsrtombs*( ), *wcscat*( ), *wcschr*( ), *wcscmp*( ), *wcscoll*( ), *wcscpy*( ), *wcscspn*( ), *wcsftime*( ), *wcslen*( ), *wcsncat*( ), *wcsncmp*( ), *wcsncpy*( ), *wcspbrk*( ), *wcsrchr*( ), *wcsspn*( ), *wcsstr*( ), *wcstod*( ), *wcstof*( ), *wcstok*( ), *wcstol*( ), *wcstold*( ), *wcstoll*( ), *wcstoul*( ), *wcstoull*( ), *wcswcs*( ), *wcswidth*( ), *wcsxfrm*( ), *wctob*( ), *wctype*( ), *wcwidth*( ), *wmemchr*( ), *wmemcmp*( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), *wprintf*( ), *wscanf*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

CHANGE HISTORY

First released in Issue 4.

Issue 5

Aligned with the ISO / IEC 9899: 1990 / Amendment 1: 1995 (E).

Issue 6

The Open Group Corrigendum U021/10 is applied. The prototypes for *wcswidth*( ) and *wcwidth*( ) are marked as extensions.

The Open Group Corrigendum U028/5 is applied, correcting the prototype for the *mbsinit*( ) function.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- Various function prototypes are updated to add the **restrict** keyword.
- The functions *vfwscanf*( ), *vswscanf*( ), *wcstof*( ), *wcstold*( ), *wcstoll*( ), and *wcstoull*( ) are added.

The type **wctype_t**, the *isw**( ), *to**( ), and *wctype*( ) functions are marked as XSI extensions.

NAME

wctype.h — wide-character classification and mapping utilities

SYNOPSIS

```
#include <wctype.h>
```

DESCRIPTION

CX Some of the functionality described on this reference page extends the ISO C standard. Applications shall define the appropriate feature test macro (see the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.2, The Compilation Environment) to enable the visibility of these symbols in this header.

The **<wctype.h>** header shall define the following types:

wint_t	As described in **<wchar.h>**.
wctrans_t	A scalar type that can hold values which represent locale-specific character mappings.
wctype_t	As described in **<wchar.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided. |

```
int        iswalnum(wint_t);
int        iswalpha(wint_t);
int        iswblank(wint_t);
int        iswcntrl(wint_t);
int        iswdigit(wint_t);
int        iswgraph(wint_t);
int        iswlower(wint_t);
int        iswprint(wint_t);
int        iswpunct(wint_t);
int        iswspace(wint_t);
int        iswupper(wint_t);
int        iswxdigit(wint_t);
int        iswctype(wint_t, wctype_t);
wint_t     towctrans(wint_t, wctrans_t);
wint_t     towlower(wint_t);
wint_t     towupper(wint_t);
wctrans_t wctrans(const char *);
wctype_t  wctype(const char *);
```

The **<wctype.h>** header shall define the following macro name:

WEOF	Constant expression of type **wint_t** that is returned by several MSE functions to indicate end-of-file.

For all functions described in this header that accept an argument of type **wint_t**, the value is representable as a **wchar_t** or equals the value of WEOF. If this argument has any other value, the behavior is undefined.

The behavior of these functions shall be affected by the *LC_CTYPE* category of the current locale.

CX Inclusion of the **<wctype.h>** header may make visible all symbols from the headers **<ctype.h>**, **<stdio.h>**, **<stdarg.h>**, **<stdlib.h>**, **<string.h>**, **<stddef.h>**, **<time.h>**, and **<wchar.h>**.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<locale.h>, **<wchar.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *iswalnum*( ), *iswalpha*( ), *iswblank*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *towctrans*( ), *towlower*( ), *towupper*( ), *wctrans*( ), *wctype*( )

CHANGE HISTORY

First released in Issue 5. Derived from the ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

Issue 6

The *iswblank*( ) function is added for alignment with the ISO/IEC 9899: 1999 standard.

NAME

wordexp.h — word-expansion types

SYNOPSIS

```
#include <wordexp.h>
```

DESCRIPTION

The **<wordexp.h>** header shall define the structures and symbolic constants used by the *wordexp*( ) and *wordfree*( ) functions.

The structure type **wordexp_t** shall contain at least the following members:

`size_t`	`we_wordc`	Count of words matched by *words*.
`char`	`**we_wordv`	Pointer to list of expanded words.
`size_t`	`we_offs`	Slots to reserve at the beginning of *we_wordv*.

The *flags* argument to the *wordexp*( ) function shall be the bitwise-inclusive OR of the following flags:

WRDE_APPEND	Append words to those previously generated.
WRDE_DOOFFS	Number of null pointers to prepend to *we_wordv*.
WRDE_NOCMD	Fail if command substitution is requested.
WRDE_REUSE	The *pwordexp* argument was passed to a previous successful call to *wordexp*( ), and has not been passed to *wordfree*( ). The result is the same as if the application had called *wordfree*( ) and then called *wordexp*( ) without WRDE_REUSE.
WRDE_SHOWERR	Do not redirect *stderr* to **/dev/null**.
WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

The following constants shall be defined as error return values:

WRDE_BADCHAR	One of the unquoted characters—<newline>, `'|'`, `'&'`, `';'`, `'<'`, `'>'`, `'('`, `')'`, `'{'`, `'}'`—appears in *words* in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.
WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in flags.
WRDE_NOSPACE	Attempt to allocate memory failed.
OB XSI WRDE_NOSYS	Reserved.
WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

The **<wordexp.h>** header shall define the following type:

XSI **size_t**	As described in **<stddef.h>**.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int  wordexp(const char *restrict, wordexp_t *restrict, int);
void wordfree(wordexp_t *);
```

The implementation may define additional macros or constants using names beginning with WRDE_.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

<stddef.h>, the System Interfaces volume of IEEE Std 1003.1-200x, *wordexp*( ), the Shell and Utilities volume of IEEE Std 1003.1-200x |

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 6

The **restrict** keyword is added to the prototype for *wordexp*( ).

The WRDE_NOSYS constant is marked obsolescent.

Index

Chapter 1

Introduction

1.1 Scope

The scope of IEEE Std 1003.1-200x is described in the Base Definitions volume of IEEE Std 1003.1-200x.

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

1.3 Normative References

Normative references for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.4 Change History

Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-200x, and in the CHANGE HISTORY section of reference pages.

1.5 Terminology

This section appears in the Base Definitions volume of IEEE Std 1003.1-200x, but is repeated here for convenience:

For the purposes of IEEE Std 1003.1-200x, the following terminology definitions apply:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to IEEE Std 1003.1-200x. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by IEEE Std 1003.1-200x but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to IEEE Std 1003.1-200x. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

shall

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is mandatory. An application can rely on the existence of the feature or behavior.

For an application or user, describes a behavior that is mandatory.

should

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by IEEE Std 1003.1-200x which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by IEEE Std 1003.1-200x which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.6 Definitions

Concepts and definitions are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.7 Relationship to Other Formal Standards

Great care has been taken to ensure that this volume of IEEE Std 1003.1-200x is fully aligned with the following standards:

ISO C (1999)
ISO / IEC 9899: 1999, Programming Languages — C.

Parts of the ISO / IEC 9899: 1999 standard (hereinafter referred to as the ISO C standard) are referenced to describe requirements also mandated by this volume of IEEE Std 1003.1-200x. Some functions and headers included within this volume of IEEE Std 1003.1-200x have a version in the ISO C standard; in this case CX markings are added as appropriate to show where the ISO C standard has been extended (see Section 1.8.1). Any conflict between this volume of IEEE Std 1003.1-200x and the ISO C standard is unintentional.

This volume of IEEE Std 1003.1-200x also allows, but does not require, mathematics functions to support IEEE Std 754-1985 and IEEE Std 854-1987.

1.8 Portability

Some of the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x and functions in the System Interfaces volume of IEEE Std 1003.1-200x describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance).

Where optional, enhanced, or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the option, extension, or warning (see Section 1.8.1). For maximum portability, an application should avoid such functionality.

1.8.1 Codes

Margin codes and their meanings are listed in the Base Definitions volume of IEEE Std 1003.1-200x, but are repeated here for convenience:

ADV Advisory Information
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

AIO Asynchronous Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the AIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the AIO margin legend.

BAR Barriers
The functionality described is optional. The functionality described is also an extension to the

ISO C standard.

Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the BAR margin legend.

BE Batch Environment Services and Utilities

The functionality described is optional.

Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the BE margin legend.

CD C-Language Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the CD margin legend.

CPT Process CPU-Time Clocks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CPT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CPT margin legend.

CS Clock Selection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CS margin legend.

CX Extension to the ISO C standard

The functionality described is an extension to the ISO C standard. Application writers may make use of an extension as it is supported on all IEEE Std 1003.1-200x-conforming systems.

With each function or header from the ISO C standard, a statement to the effect that ‘‘any conflict is unintentional’’ is included. That is intended to refer to a direct conflict. IEEE Std 1003.1-200x acts in part as a profile of the ISO C standard, and it may choose to further constrain behaviors allowed to vary by the ISO C standard. Such limitations are not considered conflicts.

FD FORTRAN Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the FD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FD margin legend.

FR FORTRAN Runtime Utilities

The functionality described is optional.

Where applicable, utilities are marked with the FR margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FR margin legend.

FSC File Synchronization

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the FSC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the FSC margin legend.

IP6 IPV6

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the IP6 margin legend.

MC1 Advisory Information and either Memory Mapped Files or Shared Memory Objects

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC1 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code Notation.

MC2 Memory Mapped Files, Shared Memory Objects, or Memory Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC2 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC2 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code Notation.

MF Memory Mapped Files

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MF margin legend.

ML Process Memory Locking

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ML margin legend.

MLR Range Memory Locking

The functionality described is optional. The functionality described is also an extension to the

ISO C standard.

Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MLR margin legend.

MON Monotonic Clock
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MON margin legend.

MPR Memory Protection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MPR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MPR margin legend.

MSG Message Passing
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MSG margin legend.

MX IEC 60559 Floating-Point Option
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MX margin legend.

OB Obsolescent
The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.

OF Output Format Incompletely Specified
The functionality described is an XSI extension. The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OH Optional Header
In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std 1003.1-200x an included header is marked as in the following example:

OH
```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

This indicates that the marked header is not required on XSI-conformant systems.

PIO

Prioritized Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.

PS

Process Scheduling

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PS margin legend.

RS

Raw Sockets

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RS margin legend.

RTS

Realtime Signals Extension

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.

SD

Software Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.

SEM

Semaphores

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.

SHM

Shared Memory Objects

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SHM margin legend.

SIO

Synchronized Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SIO margin legend.

SPI Spin Locks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPI margin legend.

SPN Spawn

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPN margin legend.

SS Process Sporadic Server

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SS margin legend.

TCT Thread CPU-Time Clocks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TCT margin legend.

TEF Trace Event Filter

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TEF margin legend.

THR Threads

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the THR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the THR margin legend.

TMO Timeouts

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMO margin legend.

TMR Timers

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMR margin legend.

TPI Thread Priority Inheritance

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPI margin legend.

TPP Thread Priority Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPP margin legend.

TPS Thread Execution Scheduling

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPS margin legend.

TRC Trace

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRC margin legend.

TRI Trace Inherit

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRI margin legend.

TRL Trace Log

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRL margin legend.

TSA Thread Stack Address Attribute

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSA margin legend.

TSF Thread-Safe Functions
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSF margin legend.

TSH Thread Process-Shared Synchronization
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSH margin legend.

TSP Thread Sporadic Server
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSP margin legend.

TSS Thread Stack Address Size
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSS margin legend.

TYM Typed Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TYM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TYM margin legend. |

UP User Portability Utilities
The functionality described is optional.

Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the UP margin legend.

XSI Extension
The functionality described is an XSI extension. Functionality marked XSI is also an extension to the ISO C standard. Application writers may confidently make use of an extension on all systems supporting the X/Open System Interfaces Extension.

If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that reference page is an extension. See the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.439, XSI.

XSR XSI STREAMS

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the XSR margin legend.

1.9 Format of Entries

The entries in Chapter 3 are based on a common format as follows. The only sections relating to conformance are the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS sections.

NAME

This section gives the name or names of the entry and briefly states its purpose.

SYNOPSIS

This section summarizes the use of the entry being described. If it is necessary to include a header to use this function, the names of such headers are shown, for example:

```
#include <stdio.h>
```

DESCRIPTION

This section describes the functionality of the function or header.

RETURN VALUE

This section indicates the possible return values, if any.

If the implementation can detect errors, ‘‘successful completion’’ means that no error has been detected during execution of the function. If the implementation does detect an error, the error is indicated.

For functions where no errors are defined, ‘‘successful completion’’ means that if the implementation checks for errors, no error has been detected. If the implementation can detect errors, and an error is detected, the indicated return value is returned and *errno* may be set.

ERRORS

This section gives the symbolic names of the error values returned by a function or stored into a variable accessed through the symbol *errno* if an error occurs.

‘‘No errors are defined’’ means that error values returned by a function or stored into a variable accessed through the symbol *errno*, if any, depend on the implementation.

EXAMPLES

This section is non-normative.

This section gives examples of usage, where appropriate. In the event of conflict between an example and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

APPLICATION USAGE

This section is non-normative.

This section gives warnings and advice to application writers about the entry. In the event of conflict between warnings and advice and a normative part of this volume of IEEE Std 1003.1-200x, the normative material is to be taken as correct.

RATIONALE

This section is non-normative.

This section contains historical information concerning the contents of this volume of IEEE Std 1003.1-200x and why features were included or discarded by the standard developers.

FUTURE DIRECTIONS

This section is non-normative.

This section provides comments which should be used as a guide to current thinking; there is not necessarily a commitment to adopt these future directions.

SEE ALSO

This section is non-normative.

This section gives references to related information.

CHANGE HISTORY

This section is non-normative.

This section shows the derivation of the entry and any significant changes that have been made to it.

Chapter 2

General Information

This chapter covers information that is relevant to all the functions specified in Chapter 3 and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

2.1 Use and Implementation of Functions

Each of the following statements shall apply unless explicitly stated otherwise in the detailed descriptions that follow:

1. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a null pointer), the behavior is undefined.
2. Any function declared in a header may also be implemented as a macro defined in the header, so a function should not be declared explicitly if its header is included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a function even if it is also defined as a macro. The use of the C-language **#undef** construct to remove any such macro definition shall also ensure that an actual function is referred to.
3. Any invocation of a function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.
4. Provided that a function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, and use it without including its associated header.
5. If a function that accepts a variable number of arguments is not declared (explicitly or by including its associated header), the behavior is undefined.

2.2 The Compilation Environment

2.2.1 POSIX.1 Symbols

Certain symbols in this volume of IEEE Std 1003.1-200x are defined in headers (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers). Some of those headers could also define symbols other than those defined by IEEE Std 1003.1-200x, potentially conflicting with symbols used by the application. Also, IEEE Std 1003.1-200x defines symbols that are not permitted by other standards to appear in those headers without some control on the visibility of those symbols.

Symbols called ‘‘feature test macros’’ are used to control the visibility of symbols that might be included in a header. Implementations, future versions of IEEE Std 1003.1-200x, and other standards may define additional feature test macros.

In the compilation of an application that **#define**s a feature test macro specified by IEEE Std 1003.1-200x, no header defined by IEEE Std 1003.1-200x shall be included prior to the definition of the feature test macro. This restriction also applies to any implementation-provided header in which these feature test macros are used. If the definition of the macro does not precede the **#include**, the result is undefined.

Feature test macros shall begin with the underscore character ('_').

2.2.1.1 The _POSIX_C_SOURCE Feature Test Macro

A POSIX-conforming application should ensure that the feature test macro _POSIX_C_SOURCE is defined before inclusion of any header.

When an application includes a header described by IEEE Std 1003.1-200x, and when this feature test macro is defined to have the value 200xxxL:

1. All symbols required by IEEE Std 1003.1-200x to appear when the header is included shall be made visible.
2. Symbols that are explicitly permitted, but not required, by IEEE Std 1003.1-200x to appear in that header (including those in reserved name spaces) may be made visible.
3. Additional symbols not required or explicitly permitted by IEEE Std 1003.1-200x to be in that header shall not be made visible, except when enabled by another feature test macro.

Identifiers in IEEE Std 1003.1-200x may only be undefined using the **#undef** directive as described in Section 2.1 (on page 463) or Section 2.2.2. These **#undef** directives shall follow all **#include** directives of any header in IEEE Std 1003.1-200x.

Note: The POSIX.1-1990 standard specified a macro called _POSIX_SOURCE. This has been superseded by _POSIX_C_SOURCE.

2.2.1.2 The _XOPEN_SOURCE Feature Test Macro

XSI An XSI-conforming application should ensure that the feature test macro _XOPEN_SOURCE is defined with the value 600 before inclusion of any header. This is needed to enable the functionality described in Section 2.2.1.1 and in addition to enable the X/Open System Interfaces Extension.

Since this volume of IEEE Std 1003.1-200x is aligned with the ISO C standard, and since all functionality enabled by _POSIX_C_SOURCE set equal to 200xxxL is enabled by _XOPEN_SOURCE set equal to 600, there should be no need to define _POSIX_C_SOURCE if _XOPEN_SOURCE is so defined. Therefore, if _XOPEN_SOURCE is set equal to 600 and _POSIX_C_SOURCE is set equal to 200xxxL, the behavior is the same as if only _XOPEN_SOURCE is defined and set equal to 600. However, should _POSIX_C_SOURCE be set to a value greater than 200xxxL, the behavior is unspecified.

2.2.2 The Name Space

All identifiers in this volume of IEEE Std 1003.1-200x, except *environ*, are defined in at least one of the headers, as shown in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, XSI Headers. When _XOPEN_SOURCE or _POSIX_C_SOURCE is defined, each header defines or declares some identifiers, potentially conflicting with identifiers used by the application. The set of identifiers visible to the application consists of precisely those identifiers from the header pages of the included headers, as well as additional identifiers reserved for the implementation. In addition, some headers may make visible identifiers from other headers as indicated on the relevant header pages.

Implementations may also add members to a structure or union without controlling the visibility of those members with a feature test macro, as long as a user-defined macro with the same name cannot interfere with the correct interpretation of the program. The identifiers reserved for use by the implementation are described below:

1. Each identifier with external linkage described in the header section is reserved for use as an identifier with external linkage if the header is included.
2. Each macro described in the header section is reserved for any use if the header is included.
3. Each identifier with file scope described in the header section is reserved for use as an identifier with file scope in the same name space if the header is included.

The prefixes posix_, POSIX_, and _POSIX_ are reserved for use by IEEE Std 1003.1-200x and other POSIX standards. Implementations may add symbols to the headers shown in the following table, provided the identifiers for those symbols begin with the corresponding reserved prefixes in the following table, and do not use the reserved prefixes posix_, POSIX_, or _POSIX_.

	Header	Prefix	Suffix	Complete Name
AIO	**<aio.h>**	aio_, lio_, AIO_, LIO_		
	<arpa/inet.h>	in_, inet_		
	<ctype.h>	to[a-z], is[a-z]		
	<dirent.h>	d_		
	<errno.h>	E[0-9], E[A-Z]		
	<fcntl.h>	l_		
	<glob.h>	gl_		
	<grp.h>	gr_		
	<inttypes.h>			int[0-9a-z_]*_t, uint[0-9a-z_]*_t
	<limits.h>		_MAX, _MIN	
	<locale.h>	LC_[A-Z]		
MSG	**<mqueue.h>**	mq_, MQ_		
XSI	**<ndbm.h>**	dbm_		
	<netdb.h>	h_, n_, p_, s_		
	<net/if.h>	if_		
	<netinet/in.h>	in_, ip_, s_, sin_		
IP6		in6_, s6_, sin6_		
XSI	**<poll.h>**	pd_, ph_, ps_		
	<pthread.h>	pthread_, PTHREAD_		
	<pwd.h>	pw_		
	<regex.h>	re_, rm_		
PS	**<sched.h>**	sched_, SCHED_		
SEM	**<semaphore.h>**	sem_, SEM_		
	<signal.h>	sa_, uc_, SIG[A-Z], SIG_[A-Z]		
XSI		ss_, sv_		
RTS		si_, SI_, sigev_, SIGEV_, sival_		
XSI	**<stropts.h>**	bi_, ic_, l_, sl_, str_		
	<stdint.h>			int[0-9a-z_]*_t, uint[0-9a-z_]*_t
	<stdlib.h>	str[a-z]		
	<string.h>	str[a-z], mem[a-z], wcs[a-z]		
XSI	**<sys/ipc.h>**	ipc_		key, pad, seq
MF	**<sys/mman.h>**	shm_, MAP_, MCL_, MS_, PROT_		
XSI	**<sys/msg.h>**	msg		msg
XSI	**<sys/resource.h>**	rlim_, ru_		
	<sys/select.h>	fd_, fds_, FD_		
XSI	**<sys/sem.h>**	sem		sem
XSI	**<sys/shm.h>**	shm		
	<sys/socket.h>	ss_, sa_, if_, ifc_, ifru_, infu_, ifra_, msg_, cmsg_, l_		
	<sys/stat.h>	st_		
XSI	**<sys/statvfs.h>**	f_		
	<sys/time.h>	fds_, it_, tv_, FD_		
	<sys/times.h>	tms_		

	Header	Prefix	Suffix	Complete Name
XSI	**<sys/uio.h>**	iov_		UIO_MAXIOV
	<sys/un.h>	sun_		
	<sys/utsname.h>	uts_		
XSI	**<sys/wait.h>**	si_, W[A-Z], P_		
	<termios.h>	c_		
	<time.h>	tm_		
TMR		clock_, timer_, it_, tv_, CLOCK_, TIMER_		
XSI	**<ucontext.h>**	uc_, ss_		
XSI	**<ulimit.h>**	UL_		
	<utime.h>	utim_		
XSI	**<utmpx.h>**	ut_	_LVL, _TIME, _PROCESS	
	<wchar.h>	wcs[a-z]		
	<wctype.h>	is[a-z], to[a-z]		
	<wordexp.h>	we_		
	ANY header	POSIX_, _POSIX_, posix_	_t	

Note: The notation [A–Z] indicates any uppercase letter in the portable character set. The notation [a–z] indicates any lowercase letter in the portable character set. Commas and spaces in the lists of prefixes and complete names in the above table are not part of any prefix or complete name.

If any header in the following table is included, macros with the prefixes shown may be defined. After the last inclusion of a given header, an application may use identifiers with the corresponding prefixes for its own purpose, provided their use is preceded by a **#undef** of the corresponding macro.

	Header	Prefix
XSI	**<dlfcn.h>**	RTLD_
	<fcntl.h>	F_, O_, S_
XSI	**<fmtmsg.h>**	MM_
	<fnmatch.h>	FNM_
XSI	**<ftw.h>**	FTW
	<glob.h>	GLOB_
	<inttypes.h>	PRI[a-z], SCN[a-z]
XSI	**<ndbm.h>**	DBM_
	<net/if.h>	IF_
	<netinet/in.h>	IMPLINK_, IN_, INADDR_, IP_, IPPORT_, IPPROTO_, SOCK_
IP6		IPV6_, IN6_
	<netinet/tcp.h>	TCP_
XSI	**<nl_types.h>**	NL_
XSI	**<poll.h>**	POLL
	<regex.h>	REG_
	<signal.h>	SA_, SIG_[0-9a-z_],
XSI		BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, SS_, SV_, TRAP_
	stdint.h	INT[0-9A-Z_]_MIN, INT[0-9A-Z_]_MAX, INT[0-9A-Z_]_C UINT[0-9A-Z_]_MIN, UINT[0-9A-Z_]_MAX, UINT[0-9A-Z_]_C
XSI	**<stropts.h>**	FLUSH[A-Z], I_, M_, MUXID_R[A-Z], S_, SND[A-Z], STR
XSI	**<syslog.h>**	LOG_
XSI	**<sys/ipc.h>**	IPC_
XSI	**<sys/mman.h>**	PROT_, MAP_, MS_
XSI	**<sys/msg.h>**	MSG[A-Z]
XSI	**<sys/resource.h>**	PRIO_, RLIM_, RLIMIT_, RUSAGE_
XSI	**<sys/sem.h>**	SEM_
XSI	**<sys/shm.h>**	SHM[A-Z], SHM_[A-Z]
XSI	**<sys/socket.h>**	AF_, CMSG_, MSG_, PF_, SCM_, SHUT_, SO
	<sys/stat.h>	S_
XSI	**<sys/statvfs.h>**	ST_
XSI	**<sys/time.h>**	FD_, ITIMER_
XSI	**<sys/uio.h>**	IOV_
XSI	**<sys/wait.h>**	BUS_, CLD_, FPE_, ILL_, POLL_, SEGV_, SI_, TRAP_
	<termios.h>	V, I, O, TC, B[0-9] (See below.)
	<wordexp.h>	WRDE_

Note: The notation [0–9] indicates any digit. The notation [A–Z] indicates any uppercase letter in the portable character set. The notation [0–9a–z_] indicates any digit, any lowercase letter in the portable character set, or underscore.

The following reserved names are used as exact matches for **<termios.h>**:

XSI
CBAUD
DEFECHO
ECHOCTL
ECHOKE
ECHOPRT
EXTA
EXTB
FLUSHO
LOBLK
PENDIN
SWTCH
VDISCARD
VDSUSP
VLNEXT
VREPRINT
VSTATUS
VWERASE

The following identifiers are reserved regardless of the inclusion of headers:

1. All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use by the implementation.
2. All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.
3. All identifiers in the table below are reserved for use as identifiers with external linkage. Some of these identifiers do not appear in this volume of IEEE Std 1003.1-200x, but are reserved for future use by the ISO C standard.

_Exit
abort
abs
acos
acosf
acosh
acoshf
acoshl
acosl
acosl
asctime
asin
asinf
asinh
asinhf
asinhl
asinl
asinl
atan
atan2
atan2f
atan2l
atanf
atanf
atanh
atanh
atanhf
atanhl
atanl
atanl
atexit
atof
atoi
atol
atoll
bsearch
cabs
cabsf
cabsl
cacos
cexp
cexpf
cexpl
cimag
cimagf
cimagl
clearerr
clock
clog
clogf
clogl
conj
conjf
conjl
copysign
copysignf
copysignl
cos
cosf
cosh
coshf
coshl
cosl
cpow
cpowf
cpowl
cproj
cprojf
cprojl
creal
crealf
creall
csin
csinf
csinh
csinhf
csinhl
csinl
csqrt
csqrtf
fesetexceptflag
fesetround
fetestexcept
feupdateenv
fflush
fgetc
fgetpos
fgets
fgetwc
fgetws
floor
floorf
floorl
fma
fmaf
fmal
fmax
fmaxf
fmaxl
fmin
fminf
fminl
fmod
fmodf
fmodl
fopen
fprintf
fputc
fputs
fputwc
fputws
fread
free
freopen
frexp
frexpf
frexpl
fscanf
fseek
fsetpos
localtime
log
log10
log10f
log10l
log1p
log1pf
log1pl
log2
log2f
log2l
logb
logbf
logbl
logf
logl
longjmp
lrint
lrintf
lrintl
lround
lroundf
lroundl
malloc
mblen
mbrlen
mbrtowc
mbsinit
mbsrtowcs
mbstowcs
mbtowc
mem[a-z]*
mktime
modf
modff
modfl
nan
nanf
nanl
nearbyint
scalbn
scalbnf
scalbnl
scanf
setbuf
setjmp
setlocale
setvbuf
signal
sin
sinf
sinh
sinhf
sinhl
sinl
sprintf
sqrt
sqrtf
sqrtl
srand
sscanf
str[a-z]*
strtof
strtoimax
strtold
strtoll
strtoull
strtoumax
swprintf
swscanf
system
tan
tanf
tanh
tanhf
tanhl
tanl
tgamma
tgammaf
tgammal

cacosf
cacosh
cacoshf
cacoshl
cacosl
calloc
carg
cargf
cargl
casin
casinf
casinh
casinhf
casinhl
casinl
catan
catanf
catanh
catanh
catanhf
catanhf
catanhl
catanhl
catanl
cbrt
cbrtf
cbrtl
ccos
ccosf
ccosh
ccoshf
ccoshl
ccosl
ceil
ceilf
ceilf
ceill
ceill
csqrtl
ctan
ctanf
ctanl
ctime
difftime
div
erfcf
erfcl
erff
erfl
errno
exit
exp
exp2
exp2f
exp2l
expf
expl
expm1
expm1f
expm1l
fabs
fabsf
fabsl
fclose
fdim
fdimf
fdiml
feclearexcept
fegetenv
fegetexceptflag
fegetround
feholdexcept
feof
feraiseexcept
ferror
fesetenv
ftell
fwide
fwprintf
fwrite
fwscanf
getc
getchar
getenv
gets
getwc
getwchar
gmtime
hypotf
hypotl
ilogb
ilogbf
ilogbl
imaxabs
imaxdiv
is[a-z]*
isblank
iswblank
labs
ldexp
ldexpf
ldexpl
ldiv
ldiv
lgammaf
lgammal
llabs
llrint
llrintf
llrintl
llround
llroundf
llroundl
localeconv
nearbyintf
nearbyintl
nextafterf
nextafterl
nexttoward
nexttowardf
nexttowardl
perror
pow
powf
powl
printf
putc
putchar
puts
putwc
putwchar
qsort
raise
rand
realloc
remainderf
remainderl
remove
remquo
remquof
remquol
rename
rewind
rint
rintf
rintl
round
roundf
roundl
scalbln
scalblnf
scalblnl
time
tmpfile
tmpnam
to[a-z]*
trunc
truncf
truncl
ungetc
ungetwc
va_end
vfprintf
vfscanf
vfwprintf
vfwscanf
vprintf
vscanf
vsprintf
vsscanf
vswprintf
vswscanf
vwprintf
vwscanf
wcrtomb
wcs[a-z]*
wcstof
wcstoimax
wcstold
wcstoll
wcstoull
wcstoumax
wctob
wctomb
wctrans
wctype
wcwidth
wmem[a-z]*
wprintf
wscanf

Note: The notation [a–z] indicates any lowercase letter in the portable character set. The notation '*' indicates any combination of digits, letters in the portable character set, or underscore.

4. All functions and external identifiers defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers are reserved for use as identifiers with external linkage.

5. All the identifiers defined in this volume of IEEE Std 1003.1-200x that have external linkage are always reserved for use as identifiers with external linkage.

No other identifiers are reserved.

Applications shall not declare or define identifiers with the same name as an identifier reserved in the same context. Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names shall not be defined by an application if any associated header is included.

Except that the effect of each inclusion of **<assert.h>** depends on the definition of NDEBUG, headers may be included in any order, and each may be included more than once in a given scope, with no difference in effect from that of being included only once.

If used, the application shall ensure that a header is included outside of any external declaration or definition, and it shall be first included before the first reference to any type or macro it defines, or to any function or object it declares. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. Prior to the inclusion of a header, the application shall not define any macros with names lexically identical to symbols defined by that header.

2.3 Error Numbers

Most functions can provide an error number. The means by which each function provides its error numbers is specified in its description.

Some functions provide the error number in a variable accessed through the symbol *errno*. The symbol *errno*, defined by including the **<errno.h>** header, expands to a modifiable lvalue of type **int**. It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name *errno*, the behavior is undefined.

The value of *errno* should only be examined when it is indicated to be valid by a function's return value. No function in this volume of IEEE Std 1003.1-200x shall set *errno* to zero. For each thread of a process, the value of *errno* shall not be affected by function calls or assignments to *errno* by other threads.

Some functions return an error number directly as the function value. These functions return a value of zero to indicate success.

If more than one error occurs in processing a function call, any one of the possible errors may be returned, as the order of detection is undefined.

Implementations may support additional errors not included in this list, may generate errors included in this list under circumstances other than those described here, or may contain extensions or limitations that prevent some errors from occurring. The ERRORS section on each reference page specifies whether an error shall be returned, or whether it may be returned. Implementations shall not generate a different error number from the ones described here for error conditions described in this volume of IEEE Std 1003.1-200x, but may generate additional errors unless explicitly disallowed for a particular function.

Each implementation shall document, in the conformance document, situations in which each of the optional conditions defined in IEEE Std 1003.1-200x is detected. The conformance document may also contain statements that one or more of the optional error conditions are not detected.

For functions under the Threads option for which [EINTR] is not listed as a possible error condition in this volume of IEEE Std 1003.1-200x, an implementation shall not return an error code of [EINTR].

The following symbolic names identify the possible error numbers, in the context of the functions specifically defined in this volume of IEEE Std 1003.1-200x; these general descriptions are more precisely defined in the ERRORS sections of the functions that return them. Only these symbolic names should be used in programs, since the actual value of the error number is unspecified. All values listed in this section shall be unique integer constant expressions with type **int** suitable for use in **#if** preprocessing directives, except as noted below. The values for all these names shall be found in the **<errno.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x. The actual values are unspecified by this volume of IEEE Std 1003.1-200x.

[E2BIG]

Argument list too long. The sum of the number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.

or:

Lack of space in an output buffer.

or:

Argument is greater than the system-imposed maximum.

[EACCES]

Permission denied. An attempt was made to access a file in a way forbidden by its file access permissions.

[EADDRINUSE]

Address in use. The specified address is in use.

[EADDRNOTAVAIL]

Address not available. The specified address is not available from the local system.

[EAFNOSUPPORT]

Address family not supported. The implementation does not support the specified address family, or the specified address is not a valid address for the address family of the specified socket.

[EAGAIN]

Resource temporarily unavailable. This is a temporary condition and later calls to the same routine may complete normally.

[EALREADY]

Connection already in progress. A connection request is already in progress for the specified socket.

[EBADF]

Bad file descriptor. A file descriptor argument is out of range, refers to no open file, or a read (write) request is made to a file that is only open for writing (reading).

[EBADMSG]

XSR Bad message. During a *read*(), *getmsg*(), *getpmsg*(), or *ioctl*() I_RECVFD request to a STREAMS device, a message arrived at the head of the STREAM that is inappropriate for the function receiving the message.

read()	Message waiting to be read on a STREAM is not a data message.
getmsg() or *getpmsg*()	A file descriptor was received instead of a control message.
ioctl()	Control or data information was received instead of a file descriptor when I_RECVFD was specified.

or:

Bad Message. The implementation has detected a corrupted message.

[EBUSY]
Resource busy. An attempt was made to make use of a system resource that is not currently available, as it is being used by another process in a manner that would have conflicted with the request being made by this process.

[ECANCELED]
Operation canceled. The associated asynchronous operation was canceled before completion.

[ECHILD]
No child process. A *wait*( ) or *waitpid*( ) function was executed by a process that had no existing or unwaited-for child process.

[ECONNABORTED]
Connection aborted. The connection has been aborted.

[ECONNREFUSED]
Connection refused. An attempt to connect to a socket was refused because there was no process listening or because the queue of connection requests was full and the underlying protocol does not support retransmissions.

[ECONNRESET]
Connection reset. The connection was forcibly closed by the peer.

[EDEADLK]
Resource deadlock would occur. An attempt was made to lock a system resource that would have resulted in a deadlock situation.

[EDESTADDRREQ]
Destination address required. No bind address was established.

[EDOM]
Domain error. An input argument is outside the defined domain of the mathematical function (defined in the ISO C standard).

[EDQUOT]
Reserved.

[EEXIST]
File exists. An existing file was mentioned in an inappropriate context; for example, as a new link name in the *link*( ) function.

[EFAULT]
Bad address. The system detected an invalid address in attempting to use an argument of a call. The reliable detection of this error cannot be guaranteed, and when not detected may result in the generation of a signal, indicating an address violation, which is sent to the process.

[EFBIG]
File too large. The size of a file would exceed the maximum file size of an implementation or offset maximum established in the corresponding file description.

[EHOSTUNREACH]
Host is unreachable. The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

[EIDRM]
Identifier removed. Returned during XSI interprocess communication if an identifier has been removed from the system.

[EILSEQ]
Illegal byte sequence. A wide-character code has been detected that does not correspond to a valid character, or a byte sequence does not form a valid wide-character code (defined in the ISO C standard).

[EINPROGRESS]
Operation in progress. This code is used to indicate that an asynchronous operation has not yet completed.

or:

O_NONBLOCK is set for the socket file descriptor and the connection cannot be immediately established.

[EINTR]
Interrupted function call. An asynchronous signal was caught by the process during the execution of an interruptible function. If the signal handler performs a normal return, the interrupted function call may return this condition (see the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**).

[EINVAL]
Invalid argument. Some invalid argument was supplied; for example, specifying an undefined signal in a *signal*( ) function or a *kill*( ) function.

[EIO]
Input/output error. Some physical input or output error has occurred. This error may be reported on a subsequent operation on the same file descriptor. Any other error-causing operation on the same file descriptor may cause the [EIO] error indication to be lost.

[EISCONN]
Socket is connected. The specified socket is already connected.

[EISDIR]
Is a directory. An attempt was made to open a directory with write mode specified.

[ELOOP]
Symbolic link loop. A loop exists in symbolic links encountered during pathname resolution. This error may also be returned if more than {SYMLOOP_MAX} symbolic links are encountered during pathname resolution.

[EMFILE]
Too many open files. An attempt was made to open more than the maximum number of {OPEN_MAX} file descriptors allowed in this process.

[EMLINK]
Too many links. An attempt was made to have the link count of a single file exceed {LINK_MAX}.

[EMSGSIZE]
Message too large. A message sent on a transport provider was larger than an internal message buffer or some other network limit.

or:

Inappropriate message buffer length.

[EMULTIHOP]
Reserved.

[ENAMETOOLONG]
Filename too long. The length of a pathname exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}. This error may also occur when pathname substitution, as a result of encountering a symbolic link during pathname resolution, results in a pathname string the size of which exceeds {PATH_MAX}.

[ENETDOWN]
Network is down. The local network interface used to reach the destination is down.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
Network unreachable. No route to the network is present.

[ENFILE]
Too many files open in system. Too many files are currently open in the system. The system has reached its predefined limit for simultaneously open files and temporarily cannot accept requests to open another one.

[ENOBUFS]
No buffer space available. Insufficient buffer resources were available in the system to perform the socket operation.

XSR [ENODATA]
No message available. No message is available on the STREAM head read queue.

[ENODEV]
No such device. An attempt was made to apply an inappropriate function to a device; for example, trying to read a write-only device such as a printer.

[ENOENT]
No such file or directory. A component of a specified pathname does not exist, or the pathname is an empty string.

[ENOEXEC]
Executable file format error. A request is made to execute a file that, although it has the appropriate permissions, is not in the format required by the implementation for executable files.

[ENOLCK]
No locks available. A system-imposed limit on the number of simultaneous file and record locks has been reached and no more are currently available.

[ENOLINK]
Reserved.

[ENOMEM]
Not enough space. The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

[ENOMSG]
No message of the desired type. The message queue does not contain a message of the required type during XSI interprocess communication.

[ENOPROTOOPT]
Protocol not available. The protocol option specified to *setsockopt*( ) is not supported by the implementation.

[ENOSPC]
No space left on a device. During the *write*( ) function on a regular file or when extending a directory, there is no free space left on the device.

XSR [ENOSR]
No STREAM resources. Insufficient STREAMS memory resources are available to perform a STREAMS-related function. This is a temporary condition; it may be recovered from if other processes release resources.

XSR [ENOSTR]
Not a STREAM. A STREAM function was attempted on a file descriptor that was not associated with a STREAMS device.

[ENOSYS]
Function not implemented. An attempt was made to use a function that is not available in this implementation.

[ENOTCONN]
Socket not connected. The socket is not connected.

[ENOTDIR]
Not a directory. A component of the specified pathname exists, but it is not a directory, when a directory was expected. |

[ENOTEMPTY]
Directory not empty. A directory other than an empty directory was supplied when an empty directory was expected.

[ENOTSOCK]
Not a socket. The file descriptor does not refer to a socket.

[ENOTSUP]
Not supported. The implementation does not support this feature of the Realtime Option Group.

[ENOTTY]
Inappropriate I/O control operation. A control function has been attempted for a file or special file for which the operation is inappropriate.

[ENXIO]
No such device or address. Input or output on a special file refers to a device that does not exist, or makes a request beyond the capabilities of the device. It may also occur when, for example, a tape drive is not on-line.

[EOPNOTSUPP]
Operation not supported on socket. The type of socket (address family or protocol) does not support the requested operation.

[EOVERFLOW]
Value too large to be stored in data type. An operation was attempted which would generate a value that is outside the range of values that can be represented in the relevant data type or that are allowed for a given data item. |

[EPERM]
Operation not permitted. An attempt was made to perform an operation limited to processes with appropriate privileges or to the owner of a file or other resource.

[EPIPE]
Broken pipe. A write was attempted on a socket, pipe, or FIFO for which there is no process

to read the data.

[EPROTO]
Protocol error. Some protocol error occurred. This error is device-specific, but is generally not related to a hardware failure.

[EPROTONOSUPPORT]
Protocol not supported. The protocol is not supported by the address family, or the protocol is not supported by the implementation.

[EPROTOTYPE]
Protocol wrong type for socket. The socket type is not supported by the protocol.

[ERANGE]
Result too large or too small. The result of the function is too large (overflow) or too small (underflow) to be represented in the available space (defined in the ISO C standard).

[EROFS]
Read-only file system. An attempt was made to modify a file or directory on a file system that is read-only.

[ESPIPE]
Invalid seek. An attempt was made to access the file offset associated with a pipe or FIFO.

[ESRCH]
No such process. No process can be found corresponding to that specified by the given process ID.

[ESTALE]
Reserved.

XSR [ETIME]
STREAM *ioctl*( ) timeout. The timer set for a STREAMS *ioctl*( ) call has expired. The cause of this error is device-specific and could indicate either a hardware or software failure, or a timeout value that is too short for the specific operation. The status of the *ioctl*( ) operation is unspecified. |

[ETIMEDOUT]
Connection timed out. The connection to a remote machine has timed out. If the connection timed out during execution of the function that reported this error (as opposed to timing out prior to the function being called), it is unspecified whether the function has completed some or all of the documented behavior associated with a successful completion of the function.

or:

Operation timed out. The time limit associated with the operation was exceeded before the operation completed.

[ETXTBSY]
Text file busy. An attempt was made to execute a pure-procedure program that is currently open for writing, or an attempt has been made to open for writing a pure-procedure program that is being executed.

[EWOULDBLOCK]
Operation would block. An operation on a socket marked as non-blocking has encountered a situation such as no data available that otherwise would have caused the function to suspend execution.

A conforming implementation may assign the same values for [EWOULDBLOCK] and [EAGAIN].

[EXDEV]
Improper link. A link to a file on another file system was attempted.

2.3.1 Additional Error Numbers

Additional implementation-defined error numbers may be defined in **<errno.h>**.

2.4 Signal Concepts

2.4.1 Signal Generation and Delivery

A signal is said to be *generated* for (or sent to) a process or thread when the event that causes the signal first occurs. Examples of such events include detection of hardware faults, timer expiration, RTS signals generated via the **sigevent** structure and terminal activity, as well as invocations of the *kill*() RTS and *sigqueue*() functions. In some circumstances, the same event generates signals for multiple processes.

At the time of generation, a determination shall be made whether the signal has been generated for the process or for a specific thread within the process. Signals which are generated by some action attributable to a particular thread, such as a hardware fault, shall be generated for the thread that caused the signal to be generated. Signals that are generated in association with a process ID or process group ID or an asynchronous event, such as terminal activity, shall be generated for the process.

Each process has an action to be taken in response to each signal defined by the system (see Section 2.4.3 (on page 480)). A signal is said to be *delivered* to a process when the appropriate action for the process and signal is taken. A signal is said to be *accepted* by a process when the signal is selected and returned by one of the *sigwait*() functions.

During the time between the generation of a signal and its delivery or acceptance, the signal is said to be *pending*. Ordinarily, this interval cannot be detected by an application. However, a signal can be *blocked* from delivery to a thread. If the action associated with a blocked signal is anything other than to ignore the signal, and if that signal is generated for the thread, the signal shall remain pending until it is unblocked, it is accepted when it is selected and returned by a call to the *sigwait*() function, or the action associated with it is set to ignore the signal. Signals generated for the process shall be delivered to exactly one of those threads within the process which is in a call to a *sigwait*() function selecting that signal or has not blocked delivery of the signal. If there are no threads in a call to a *sigwait*() function selecting that signal, and if all threads within the process block delivery of the signal, the signal shall remain pending on the process until a thread calls a *sigwait*() function selecting that signal, a thread unblocks delivery of the signal, or the action associated with the signal is set to ignore the signal. If the action associated with a blocked signal is to ignore the signal and if that signal is generated for the process, it is unspecified whether the signal is discarded immediately upon generation or remains pending.

Each thread has a *signal mask* that defines the set of signals currently blocked from delivery to it. | The signal mask for a thread shall be initialized from that of its parent or creating thread, or from | the corresponding thread in the parent process if the thread was created as the result of a call to | *fork*(). The *sigaction*(), *sigprocmask*(), and *sigsuspend*() functions control the manipulation of the signal mask.

The determination of which action is taken in response to a signal is made at the time the signal is delivered, allowing for any changes since the time of generation. This determination is independent of the means by which the signal was originally generated. If a subsequent occurrence of a pending signal is generated, it is implementation-defined as to whether the RTS signal is delivered or accepted more than once in circumstances other than those in which queuing is required under the Realtime Signals Extension option. The order in which multiple, simultaneously pending signals outside the range SIGRTMIN to SIGRTMAX are delivered to or accepted by a process is unspecified.

When any stop signal (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) is generated for a process, any pending SIGCONT signals for that process shall be discarded. Conversely, when SIGCONT is generated for a process, all pending stop signals for that process shall be discarded. When SIGCONT is generated for a process that is stopped, the process shall be continued, even if the SIGCONT signal is blocked or ignored. If SIGCONT is blocked and not ignored, it shall remain pending until it is either unblocked or a stop signal is generated for the process.

An implementation shall document any condition not specified by this volume of IEEE Std 1003.1-200x under which the implementation generates signals.

2.4.2 Realtime Signal Generation and Delivery

RTS This section describes extensions to support realtime signal generation and delivery. This functionality is dependent on support of the Realtime Signals Extension option (and the rest of this section is not further shaded for this option).

Some signal-generating functions, such as high-resolution timer expiration, asynchronous I/O completion, interprocess message arrival, and the *sigqueue*( ) function, support the specification of an application-defined value, either explicitly as a parameter to the function or in a **sigevent** structure parameter. The **sigevent** structure is defined in **<signal.h>** and contains at least the following members:

Member Type	Member Name	Description
int	*sigev_notify*	Notification type.
int	*sigev_signo*	Signal number.
union sigval	*sigev_value*	Signal value.
void(*)(unsigned sigval)	*sigev_notify_function*	Notification function.
(pthread_attr_t*)	*sigev_notify_attributes*	Notification attributes.

The *sigev_notify* member specifies the notification mechanism to use when an asynchronous event occurs. This volume of IEEE Std 1003.1-200x defines the following values for the *sigev_notify* member:

SIGEV_NONE	No asynchronous notification shall be delivered when the event of interest occurs.
SIGEV_SIGNAL	The signal specified in *sigev_signo* shall be generated for the process when the event of interest occurs. If the implementation supports the Realtime Signals Extension option and if the SA_SIGINFO flag is set for that signal number, then the signal shall be queued to the process and the value specified in *sigev_value* shall be the *si_value* component of the generated signal. If SA_SIGINFO is not set for that signal number, it is unspecified whether the signal is queued and what value, if any, is sent.
SIGEV_THREAD	A notification function shall be called to perform notification.

An implementation may define additional notification mechanisms.

The *sigev_signo* member specifies the signal to be generated. The *sigev_value* member is the application-defined value to be passed to the signal-catching function at the time of the signal delivery or to be returned at signal acceptance as the *si_value* member of the **siginfo_t** structure.

The **sigval** union is defined in **<signal.h>** and contains at least the following members: |

Member Type	Member Name	Description
int	*sival_int*	Integer signal value.
void*	*sival_ptr*	Pointer signal value.

The *sival_int* member shall be used when the application-defined value is of type **int**; the *sival_ptr* member shall be used when the application-defined value is a pointer.

When a signal is generated by the *sigqueue*( ) function or any signal-generating function that supports the specification of an application-defined value, the signal shall be marked pending and, if the SA_SIGINFO flag is set for that signal, the signal shall be queued to the process along with the application-specified signal value. Multiple occurrences of signals so generated are queued in FIFO order. It is unspecified whether signals so generated are queued when the SA_SIGINFO flag is not set for that signal.

Signals generated by the *kill*( ) function or other events that cause signals to occur, such as detection of hardware faults, *alarm*( ) timer expiration, or terminal activity, and for which the implementation does not support queuing, shall have no effect on signals already queued for the same signal number.

When multiple unblocked signals, all in the range SIGRTMIN to SIGRTMAX, are pending, the behavior shall be as if the implementation delivers the pending unblocked signal with the lowest signal number within that range. No other ordering of signal delivery is specified.

If, when a pending signal is delivered, there are additional signals queued to that signal number, the signal shall remain pending. Otherwise, the pending indication shall be reset.

Multi-threaded programs can use an alternate event notification mechanism. When a notification is processed, and the *sigev_notify* member of the **sigevent** structure has the value SIGEV_THREAD, the function *sigev_notify_function* is called with parameter *sigev_value*.

The function shall be executed in an environment as if it were the *start_routine* for a newly created thread with thread attributes specified by *sigev_notify_attributes*. If *sigev_notify_attributes* is NULL, the behavior shall be as if the thread were created with the *detachstate* attribute set to PTHREAD_CREATE_DETACHED. Supplying an attributes structure with a *detachstate* attribute of PTHREAD_CREATE_JOINABLE results in undefined behavior. The signal mask of this thread is implementation-defined.

2.4.3 Signal Actions

There are three types of action that can be associated with a signal: SIG_DFL, SIG_IGN, or a pointer to a function. Initially, all signals shall be set to SIG_DFL or SIG_IGN prior to entry of the *main*( ) routine (see the *exec* functions). The actions prescribed by these values are as follows:

SIG_DFL Signal-specific default action.

The default actions for the signals defined in this volume of IEEE Std 1003.1-200x are specified under **<signal.h>**. RTS If the Realtime Signals Extension option is supported, the default actions for the realtime signals in the range SIGRTMIN to SIGRTMAX shall be to terminate the process abnormally.

If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal shall be generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag. While a process is stopped, any additional signals that are sent to the process shall not be delivered until the process is continued, except SIGKILL which always terminates the receiving process. A process that is a member of an orphaned process group shall not be allowed to stop in response to the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of these signals would stop such a process, the signal shall be discarded.

Setting a signal action to SIG_DFL for a signal that is pending, and whose default action is to ignore the signal (for example, SIGCHLD), shall cause the pending signal to be discarded, whether or not it is blocked.

RTS The default action for SIGCONT is to resume execution at the point where the process was stopped, after first handling any pending unblocked signals. If the Realtime Signals Extension option is supported, any queued values pending shall be discarded and the resources used to queue them shall be released and returned to the system for other use. When a stopped process is continued, a SIGCHLD signal may be generated for its parent process, unless the parent process has set the SA_NOCLDSTOP flag.

SIG_IGN Ignore signal.

RTS Delivery of the signal shall have no effect on the process. The behavior of a process is undefined after it ignores a SIGFPE, SIGILL, SIGSEGV, or SIGBUS signal that was not generated by *kill*( ), *sigqueue*( ), or *raise*( ).

The system shall not allow the action for the signals SIGKILL or SIGSTOP to be set to SIG_IGN.

Setting a signal action to SIG_IGN for a signal that is pending shall cause the pending signal to be discarded, whether or not it is blocked.

XSI If a process sets the action for the SIGCHLD signal to SIG_IGN, the behavior is unspecified, except as specified below.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait*( ), *waitid*( ), and *waitpid*( ) shall fail and set *errno* to [ECHILD].

RTS If the Realtime Signals Extension option is supported, any queued values pending shall be discarded and the resources used to queue them shall be released and made available to queue other signals.

pointer to a function
Catch signal.

On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process shall resume execution at the point at which it was interrupted.

If the SA_SIGINFO flag for the signal is cleared, the signal-catching function shall be entered as a C-language function call as follows:

```
void func(int signo);
```

XSI | RTS If the SA_SIGINFO flag for the signal is set, the signal-catching function shall be entered as a C-language function call as follows:

```
void func(int signo, siginfo_t *info, void *context);
```

where *func* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, and *info* is a pointer to a **siginfo_t** structure defined in **<signal.h>** containing at least the following members:

Member Type	**Member Name**	**Description**
int	*si_signo*	Signal number.
int	*si_code*	Cause of the signal.
union sigval	*si_value*	Signal value.

The *si_signo* member shall contain the signal number. This shall be the same as the *signo* parameter. The *si_code* member shall contain a code identifying the cause of the signal. The following values are defined for *si_code*:

	SI_USER	The signal was sent by the *kill*( ) function. The implementation may set *si_code* to SI_USER if the signal was sent by the *raise*( ) or *abort*( ) functions or any similar functions provided as implementation extensions.
RTS	SI_QUEUE	The signal was sent by the *sigqueue*( ) function.
RTS	SI_TIMER	The signal was generated by the expiration of a timer set by *timer_settime*( ).
RTS	SI_ASYNCIO	The signal was generated by the completion of an asynchronous I/O request.
RTS	SI_MESGQ	The signal was generated by the arrival of a message on an empty message queue.

If the signal was not generated by one of the functions or events listed above, the *si_code* shall be set to an implementation-defined value that is not equal to any of the values defined above.

RTS If the Realtime Signals Extension is supported, and *si_code* is one of SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MESGQ, then *si_value* shall contain the application-specified signal value. Otherwise, the contents of *si_value* are undefined.

The behavior of a process is undefined after it returns normally from a signal-
XSI catching function for a SIGBUS, SIGFPE, SIGILL, or SIGSEGV signal that was not
RTS generated by *kill*( ), *sigqueue*( ), or *raise*( ).

The system shall not allow a process to catch the signals SIGKILL and SIGSTOP.

If a process establishes a signal-catching function for the SIGCHLD signal while it has a terminated child process for which it has not waited, it is unspecified whether a SIGCHLD signal is generated to indicate that child process.

When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this volume of IEEE Std 1003.1-200x is unspecified if they are called from a signal-catching function.

The following table defines a set of functions that shall be either reentrant or non-interruptible by signals and shall be async-signal-safe. Therefore applications may invoke them, without restriction, from signal-catching functions:

Notes to Reviewers

This section with side shading will not appear in the final copy. - Ed.

The contents of the following tables need to be reviewed.

_Exit( )
_exit( )
access( )
aio_error( )
aio_return( )
aio_suspend( )
alarm( )
cfgetispeed( )
cfgetospeed( )
cfsetispeed( )
cfsetospeed( )
chdir( )
chmod( )
chown( )
clock_gettime( )
close( )
creat( )
dup( )
dup2( )
execle( )
execve( )
fchmod( )
fchown( )
fcntl( )
fdatasync( )
fork( )
fpathconf( )
fstat( )
fsync( )
ftruncate( )
getegid( )
geteuid( )
getgid( )
getgroups( )
getpgrp( )
getpid( )
getppid( )
getuid( )
kill( )
link( )
lseek( )
lstat( )
mkdir( )
mkfifo( )
open( )
pathconf( )
pause( )
pipe( )
posix_trace_event( )
raise( )
read( )
readlink( )
rename( )
rmdir( )
sem_post( )
setgid( )
setpgid( )
setsid( )
setuid( )
sigaction( )
sigaddset( )
sigdelset( )
sigemptyset( )
sigfillset( )
sigismember( )
sleep( )
signal( )
sigpause( )
sigpending( )
sigprocmask( )
sigqueue( )
sigset( )
sigsuspend( )
stat( )
symlink( )
sysconf( )
tcdrain( )
tcflow( )
tcflush( )
tcgetattr( )
tcgetpgrp( )
tcsendbreak( )
tcsetattr( )
tcsetpgrp( )
time( )
timer_getoverrun( )
timer_gettime( )
timer_settime( )
times( )
umask( )
uname( )
unlink( )
utime( )
wait( )
waitpid( )
write( )

All functions not in the above table are considered to be unsafe with respect to signals. In the presence of signals, all functions defined by this volume of IEEE Std 1003.1-200x shall behave as defined when called from or interrupted by a signal-catching function, with a single exception: when a signal interrupts an unsafe function and the signal-catching function calls an unsafe function, the behavior is undefined.

When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or continue, the entire process shall be terminated, stopped, or continued, respectively.

2.4.4 Signal Effects on Other Functions

Signals affect the behavior of certain functions defined by this volume of IEEE Std 1003.1-200x if delivered to a process while it is executing such a function. If the action of the signal is to terminate the process, the process shall be terminated and the function shall not return. If the action of the signal is to stop the process, the process shall stop until continued or terminated. Generation of a SIGCONT signal for the process shall cause the process to be continued, and the original function shall continue at the point the process was stopped. If the action of the signal is to invoke a signal-catching function, the signal-catching function shall be invoked; in this case the original function is said to be *interrupted* by the signal. If the signal-catching function executes a **return** statement, the behavior of the interrupted function shall be as described individually for that function, except as noted for unsafe functions. Signals that are ignored shall not affect the behavior of any function; signals that are blocked shall not affect the behavior of any function until they are unblocked and then delivered, except as specified for the *sigpending*( ) and *sigwait*( ) functions.

2.5 Standard I/O Streams

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded if necessary. If a file can support positioning requests, (such as a disk file, as opposed to a terminal), then a *file position indicator* associated with the stream is positioned at the start (byte number 0) of the file, unless the file is opened with append mode, in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file. All input takes place as if bytes were read by successive calls to *fgetc*( ); all output takes place as if bytes were written by successive calls to *fputc*( ).

When a stream is *unbuffered*, bytes are intended to appear from the source or at the destination as soon as possible; otherwise, bytes may be accumulated and transmitted as a block. When a stream is *fully buffered*, bytes are intended to be transmitted as a block when a buffer is filled. When a stream is *line buffered*, bytes are intended to be transmitted as a block when a newline byte is encountered. Furthermore, bytes are intended to be transmitted as a block when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line-buffered stream that requires the transmission of bytes. Support for these characteristics is implementation-defined, and may be affected via *setbuf*( ) and *setvbuf*( ).

A file may be disassociated from a controlling stream by *closing* the file. Output streams are flushed (any unwritten buffer contents are transmitted) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is unspecified after the associated file is closed (including the standard streams).

A file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the *main*( ) function returns to its original caller, or if the *exit*( ) function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling *abort*( ), need not close all files properly.

The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object need not necessarily serve in place of the original.

At program start-up, three streams are predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

2.5.1 Interaction of File Descriptors and Standard I/O Streams

CX This section describes the interaction of file descriptors and standard I/O streams. This functionality is an extension to the ISO C standard (and the rest of this section is not further CX shaded).

An open file description may be accessed through a file descriptor, which is created using functions such as *open*( ) or *pipe*( ), or through a stream, which is created using functions such as *fopen*( ) or *popen*( ). Either a file descriptor or a stream is called a *handle* on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by explicit user action, without affecting the underlying open file description. Some of the ways to create them include *fcntl*( ), *dup*( ), *fdopen*( ), *fileno*( ), and *fork*( ). They can be destroyed by at least *fclose*( ), *close*( ), and the *exec* functions.

A file descriptor that is never used in an operation that could affect the file offset (for example, *read*( ), *write*( ), or *lseek*( )) is not considered a handle for this discussion, but could give rise to one (for example, as a consequence of *fdopen*( ), *dup*( ), or *fork*( )). This exception does not include the file descriptor underlying a stream, whether created with *fopen*( ) or *fdopen*( ), so long as it is not used directly by the application to affect the file offset. The *read*( ) and *write*( ) functions implicitly affect the file offset; *lseek*( ) explicitly affects it.

The result of function calls involving any one handle (the *active handle*) is defined elsewhere in this volume of IEEE Std 1003.1-200x, but if two or more handles are used, and any one of them is a stream, the application shall ensure that their actions are coordinated as described below. If this is not done, the result is undefined.

A handle which is a stream is considered to be closed when either an *fclose*( ) or *freopen*( ) is executed on it (the result of *freopen*( ) is a new stream, which cannot be a handle on the same open file description as its previous value), or when the process owning that stream terminates with *exit*( ) or *abort*( ). A file descriptor is closed by *close*( ), *_exit*( ), or the *exec* functions when FD_CLOEXEC is set on that file descriptor.

For a handle to become the active handle, the application shall ensure that the actions below are performed between the last use of the handle (the current active handle) and the first use of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active file handle. (If a stream function has as an underlying function one that affects the file offset, the stream function shall be considered to affect the file offset.)

The handles need not be in the same process for these rules to apply.

Note that after a *fork*( ), two handles exist where one existed before. The application shall ensure that, if both handles can ever be accessed, they are both in a state where the other could become the active handle first. The application shall prepare for a *fork*( ) exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the *exec* functions or *_exit*( ) (not *exit*( )), the handle is never accessed in that process.)

For the first handle, the first applicable condition below applies. After the actions required below are taken, if the handle is still open, the application can close it.

- If it is a file descriptor, no action is required.

- If the only further action to be performed on any handle to this open file descriptor is to close it, no action need be taken.
- If it is a stream which is unbuffered, no action need be taken.
- If it is a stream which is line buffered, and the last byte written to the stream was a newline (that is, as if a:

  ```
  putc('\n')
  ```

  was the most recent operation on that stream), no action need be taken.
- If it is a stream which is open for writing or appending (but not also open for reading), the application shall either perform an *fflush*( ), or the stream shall be closed.
- If the stream is open for reading and it is at the end of the file (*feof*( ) is true), no action need be taken.
- If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, the application shall either perform an *fflush*( ), or the stream shall be closed.

Otherwise, the result is undefined.

For the second handle:

- If any previous active handle has been used by a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform an *lseek*( ) or *fseek*( ) (as appropriate to the type of handle) to an appropriate location.

If the active handle ceases to be accessible before the requirements on the first handle, above, have been met, the state of the open file description becomes undefined. This might occur during functions such as a *fork*( ) or *_exit*( ).

The *exec* functions make inaccessible all streams that are open at the time they are called, independent of which streams or file descriptors may be available to the new process image.

When these rules are followed, regardless of the sequence of handles used, implementations shall ensure that an application, even one consisting of several processes, shall yield correct results: no data shall be lost or duplicated when writing, and all data shall be written in order, except as requested by seeks. It is implementation-defined whether, and under what conditions, all input is seen exactly once.

If the rules above are not followed, the result is unspecified.

Each function that operates on a stream is said to have zero or more *underlying functions*. This means that the stream function shares certain traits with the underlying functions, but does not require that there be any relation between the implementations of the stream function and its underlying functions.

2.5.2 Stream Orientation and Encoding Rules

For conformance to the ISO/IEC 9899:1999 standard, the definition of a stream includes an *orientation*. After a stream is associated with an external file, but before any operations are performed on it, the stream is without orientation. Once a wide-character input/output function has been applied to a stream without orientation, the stream shall become *wide-oriented*. Similarly, once a byte input/output function has been applied to a stream without orientation, the stream shall become *byte-oriented*. Only a call to the *freopen*( ) function or the *fwide*( ) function can otherwise alter the orientation of a stream.

A successful call to *freopen*( ) shall remove any orientation. The three predefined streams *standard input*, *standard output*, and *standard error* shall be unoriented at program start-up.

Byte input/output functions cannot be applied to a wide-oriented stream, and wide-character input/output functions cannot be applied to a byte-oriented stream. The remaining stream operations shall not affect and shall not be affected by a stream's orientation, except for the following additional restrictions:

- Binary wide-oriented streams have the file positioning restrictions ascribed to both text and binary streams.
- For wide-oriented streams, after a successful call to a file-positioning function that leaves the file position indicator prior to the end-of-file, a wide-character output function can overwrite a partial character; any file contents beyond the byte(s) written are henceforth undefined.

Each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream. A successful call to *fgetpos*( ) shall store a representation of the value of this **mbstate_t** object as part of the value of the **fpos_t** object. A later successful call to *fsetpos*( ) using the same stored **fpos_t** value shall restore the value of the associated **mbstate_t** object as well as the position within the controlled stream.

Implementations that support multiple encoding rules associate an encoding rule with the stream. The encoding rule shall be determined by the setting of the *LC_CTYPE* category in the current locale at the time when the stream becomes wide-oriented. If a wide-character input/output function is applied to a byte-oriented stream, the encoding rule used is undefined. As with the stream's orientation, the encoding rule associated with a stream cannot be changed once it has been set, except by a successful call to *freopen*( ) which clears the encoding rule and resets the orientation to unoriented.

Although both text and binary wide-oriented streams are conceptually sequences of wide characters, the external file associated with a wide-oriented stream is a sequence of (possibly multi-byte) characters generalized as follows:

- Multi-byte encodings within files may contain embedded null bytes (unlike multi-byte encodings valid for use internal to the program).
- A file need not begin nor end in the initial shift state.

Moreover, the encodings used for characters may differ among files. Both the nature and choice of such encodings are implementation-defined.

The wide-character input functions read characters from the stream and convert them to wide characters as if they were read by successive calls to the *fgetwc*( ) function. Each conversion shall occur as if by a call to the *mbrtowc*( ) function, with the conversion state described by the stream's CX own **mbstate_t** object, except the encoding rule associated with the stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current locale.

The wide-character output functions convert wide characters to (possibly multi-byte) characters and write them to the stream as if they were written by successive calls to the *fputwc*( ) function. Each conversion shall occur as if by a call to the *wcrtomb*( ) function, with the conversion state CX described by the stream's own **mbstate_t** object, except the encoding rule associated with the stream is used instead of the encoding rule implied by the *LC_CTYPE* category of the current locale.

An *encoding error* shall occur if the character sequence presented to the underlying *mbrtowc*( ) function does not form a valid (generalized) character, or if the code value passed to the underlying *wcrtomb*( ) function does not correspond to a valid (generalized) character. The wide-character input/output functions and the byte input/output functions store the value of the macro [EILSEQ] in *errno* if and only if an encoding error occurs.

2.6 STREAMS

XSR STREAMS functionality is provided on implementations supporting the XSI STREAMS Option Group. This functionality is dependent on support of the XSI STREAMS option (and the rest of this section is not further shaded for this option).

STREAMS provides a uniform mechanism for implementing networking services and other character-based I/O. The STREAMS function provides direct access to protocol modules. STREAMS modules are unspecified objects. Access to STREAMS modules is provided by interfaces in IEEE Std 1003.1-200x. Creation of STREAMS modules is outside the scope of IEEE Std 1003.1-200x.

A STREAM is typically a full-duplex connection between a process and an open device or pseudo-device. However, since pipes may be STREAMS-based, a STREAM can be a full-duplex connection between two processes. The STREAM itself exists entirely within the implementation and provides a general character I/O function for processes. It optionally includes one or more intermediate processing modules that are interposed between the process end of the STREAM (STREAM head) and a device driver at the end of the STREAM (STREAM end).

STREAMS I/O is based on messages. There are three types of message:

- *Data messages* containing actual data for input or output
- *Control data* containing instructions for the STREAMS modules and underlying implementation
- Other messages, which include file descriptors

The interface between the STREAM and the rest of the implementation is provided by a set of functions at the STREAM head. When a process calls *write*( ), *writev*( ), *putmsg*( ), *putpmsg*( ), or *ioctl*( ), messages are sent down the STREAM, and *read*( ), *readv*( ), *getmsg*( ), or *getpmsg*( ) accepts data from the STREAM and passes it to a process. Data intended for the device at the downstream end of the STREAM is packaged into messages and sent downstream, while data and signals from the device are composed into messages by the device driver and sent upstream to the STREAM head.

When a STREAMS-based device is opened, a STREAM shall be created that contains the STREAM head and the STREAM end (driver). If pipes are STREAMS-based in an implementation, when a pipe is created, two STREAMS shall be created, each containing a STREAM head. Other modules are added to the STREAM using *ioctl*( ). New modules are ‘‘pushed’’ onto the STREAM one at a time in last-in, first-out (LIFO) style, as though the STREAM was a push-down stack.

Priority

Message types are classified according to their queuing priority and may be *normal* (non-priority), *priority*, or *high-priority* messages. A message belongs to a particular priority band that determines its ordering when placed on a queue. Normal messages have a priority band of 0 and shall always be placed at the end of the queue following all other messages in the queue. High-priority messages are always placed at the head of a queue, but shall be discarded if there is already a high-priority message in the queue. Their priority band shall be ignored; they are high-priority by virtue of their type. Priority messages have a priority band greater than 0. Priority messages are always placed after any messages of the same or higher priority. High-priority and priority messages are used to send control and data information outside the normal flow of control. By convention, high-priority messages shall not be affected by flow control. Normal and priority messages have separate flow controls.

Message Parts

A process may access STREAMS messages that contain a data part, control part, or both. The data part is that information which is transmitted over the communication medium and the control information is used by the local STREAMS modules. The other types of messages are used between modules and are not accessible to processes. Messages containing only a data part are accessible via *putmsg*( ), *putpmsg*( ), *getmsg*( ), *getpmsg*( ), *read*( ), *readv*( ), *write*( ), or *writev*( ). Messages containing a control part with or without a data part are accessible via calls to *putmsg*( ), *putpmsg*( ), *getmsg*( ), or *getpmsg*( ).

2.6.1 Accessing STREAMS

A process accesses STREAMS-based files using the standard functions *close*( ), *ioctl*( ), *getmsg*( ), *getpmsg*( ), *open*( ), *pipe*( ), *poll*( ), *putmsg*( ), *putpmsg*( ), *read*( ), or *write*( ). Refer to the applicable function definitions for general properties and errors.

Calls to *ioctl*( ) shall perform control functions on the STREAM associated with the file descriptor *fildes*. The control functions may be performed by the STREAM head, a STREAMS module, or the STREAMS driver for the STREAM.

STREAMS modules and drivers can detect errors, sending an error message to the STREAM head, thus causing subsequent functions to fail and set *errno* to the value specified in the message. In addition, STREAMS modules and drivers can elect to fail a particular *ioctl*( ) request alone by sending a negative acknowledgement message to the STREAM head. This shall cause just the pending *ioctl*( ) request to fail and set *errno* to the value specified in the message.

2.7 XSI Interprocess Communication

XSI This section describes extensions to support interprocess communication. This functionality is dependent on support of the XSI Extension (and the rest of this section is not further shaded for this option).

The following message passing, semaphore, and shared memory services form an XSI interprocess communication facility. Certain aspects of their operation are common, and are described below.

IPC Functions		
msgctl( )	*semctl*( )	*shmctl*( )
msgget( )	*semget*( )	*shmdt*( )
msgrcv( )	*semop*( )	*shmget*( )
msgsnd( )	*shmat*( )	

Another interprocess communication facility is provided by functions in the Realtime Option Group; see Section 2.8 (on page 491).

2.7.1 IPC General Description

Each individual shared memory segment, message queue, and semaphore set shall be identified by a unique positive integer, called, respectively, a shared memory identifier, *shmid*, a semaphore identifier, *semid*, and a message queue identifier, *msqid*. The identifiers shall be returned by calls to *shmget*( ), *semget*( ), and *msgget*( ), respectively.

Associated with each identifier is a data structure which contains data related to the operations which may be or may have been performed; see the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, **<sys/sem.h>**, and **<sys/msg.h>** for their descriptions.

Each of the data structures contains both ownership information and an **ipc_perm** structure (see the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**) which are used in conjunction to determine whether or not read/write (read/alter for semaphores) permissions should be granted to processes using the IPC facilities. The *mode* member of the **ipc_perm** structure acts as a bit field which determines the permissions.

The values of the bits are given below in octal notation.

Bit	Meaning
0400	Read by user.
0200	Write by user.
0040	Read by group.
0020	Write by group.
0004	Read by others.
0002	Write by others.

The name of the **ipc_perm** structure is *shm_perm*, *sem_perm*, or *msg_perm*, depending on which service is being used. In each case, read and write/alter permissions shall be granted to a process if one or more of the following are true ("xxx" is replaced by *shm*, *sem*, or *msg*, as appropriate):

- The process has appropriate privileges.
- The effective user ID of the process matches *xxx_perm.cuid* or *xxx_perm.uid* in the data structure associated with the IPC identifier, and the appropriate bit of the *user* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* but the effective group ID of the process matches *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, and the appropriate bit of the *group* field in *xxx_perm.mode* is set.
- The effective user ID of the process does not match *xxx_perm.cuid* or *xxx_perm.uid* and the effective group ID of the process does not match *xxx_perm.cgid* or *xxx_perm.gid* in the data structure associated with the IPC identifier, but the appropriate bit of the *other* field in *xxx_perm.mode* is set.

Otherwise, the permission shall be denied.

2.8 Realtime

This section defines functions to support the source portability of applications with realtime requirements. The presence of many of these functions is dependent on support for implementation options described in the text.

The specific functional areas included in this section and their scope include the following. Full definitions of these terms can be found in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions.

- Semaphores
- Process Memory Locking
- Memory Mapped Files and Shared Memory Objects
- Priority Scheduling
- Realtime Signal Extension
- Timers
- Interprocess Communication
- Synchronized Input and Output
- Asynchronous Input and Output

All the realtime functions defined in this volume of IEEE Std 1003.1-200x are portable, although some of the numeric parameters used by an implementation may have hardware dependencies.

2.8.1 Realtime Signals

RTS Realtime signal generation and delivery is dependent on support for the Realtime Signals Extension option.

See Section 2.4.2 (on page 479).

2.8.2 Asynchronous I/O

AIO The functionality described in this section is dependent on support of the Asynchronous Input and Output option (and the rest of this section is not further shaded for this option).

An asynchronous I/O control block structure **aiocb** is used in many asynchronous I/O functions. It is defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>** and has at least the following members:

Member Type	Member Name	Description
int	*aio_fildes*	File descriptor.
off_t	*aio_offset*	File offset.
volatile void*	*aio_buf*	Location of buffer.
size_t	*aio_nbytes*	Length of transfer.
int	*aio_reqprio*	Request priority offset.
struct sigevent	*aio_sigevent*	Signal number and value.
int	*aio_lio_opcode*	Operation to be performed.

The *aio_fildes* element is the file descriptor on which the asynchronous operation is performed.

If O_APPEND is not set for the file descriptor *aio_fildes* and if *aio_fildes* is associated with a device that is capable of seeking, then the requested operation takes place at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to the operation with an *offset* argument equal to *aio_offset* and a *whence* argument equal to SEEK_SET. If O_APPEND is set for the file descriptor, or if *aio_fildes* is associated with a device that is incapable of seeking, write operations append to the file in the same order as the calls were made, with the following exception: under implementation-defined circumstances, such as operation on a multi-processor or when requests of differing priorities are submitted at the same time, the ordering restriction may be relaxed. Since there is no way for a strictly conforming application to determine whether this relaxation applies, all strictly conforming applications which rely on ordering of output shall be written in such a way that they will operate correctly if the relaxation applies. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified. The *aio_nbytes* and *aio_buf* elements are the same as the *nbyte* and *buf* arguments defined by *read*( ) and *write*( ), respectively.

If _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, then asynchronous I/O is queued in priority order, with the priority of each asynchronous operation based on the current scheduling priority of the calling process. The *aio_reqprio* member can be used to lower (but not raise) the asynchronous I/O operation priority and is within the range zero through {AIO_PRIO_DELTA_MAX}, inclusive. Unless both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing asynchronous I/O requests is unspecified. When both _POSIX_PRIORITIZED_IO and _POSIX_PRIORITY_SCHEDULING are defined, the order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is unspecified. The priority of an asynchronous request is computed as (process scheduling priority) minus *aio_reqprio*. The priority assigned to each asynchronous I/O request is an indication of the desired order of execution of the request relative to other asynchronous I/O requests for this file. If _POSIX_PRIORITIZED_IO is defined, requests issued with the same priority to a character special file are processed by the underlying device in FIFO order; the order of processing of requests of the same priority issued to files that are not character special files is unspecified. Numerically higher priority values indicate requests of higher priority. The value of *aio_reqprio* has no effect on process scheduling priority. When prioritized asynchronous I/O requests to the same file are blocked waiting for a resource required for that I/O operation, the higher-priority I/O requests shall be granted the resource before lower-priority I/O requests are granted the resource. The relative priority of asynchronous I/O and synchronous I/O is implementation-defined. If _POSIX_PRIORITIZED_IO is defined, the implementation shall define for which files I/O prioritization is supported.

The *aio_sigevent* determines how the calling process shall be notified upon I/O completion, as specified in Section 2.4.1 (on page 478). If *aio_sigevent.sigev_notify* is SIGEV_NONE, then no signal shall be posted upon I/O completion, but the error status for the operation and the return status for the operation shall be set appropriately.

The *aio_lio_opcode* field is used only by the *lio_listio*( ) call. The *lio_listio*( ) call allows multiple asynchronous I/O operations to be submitted at a single time. The function takes as an argument an array of pointers to **aiocb** structures. Each **aiocb** structure indicates the operation to be performed (read or write) via the *aio_lio_opcode* field.

The address of the **aiocb** structure is used as a handle for retrieving the error status and return status of the asynchronous operation while it is in progress.

The **aiocb** structure and the data buffers associated with the asynchronous I/O operation are being used by the system for asynchronous I/O while, and only while, the error status of the asynchronous operation is equal to [EINPROGRESS]. Applications shall not modify the **aiocb** structure while the structure is being used by the system for asynchronous I/O.

The return status of the asynchronous operation is the number of bytes transferred by the I/O operation. If the error status is set to indicate an error completion, then the return status is set to the return value that the corresponding *read*( ), *write*( ), or *fsync*( ) call would have returned. When the error status is not equal to [EINPROGRESS], the return status shall reflect the return status of the corresponding synchronous operation.

2.8.3 Memory Management

2.8.3.1 Memory Locking

ML The functionality described in this section is dependent on support of the Process Memory Locking option (and the rest of this section is not further shaded for this option).

Range memory locking operations are defined in terms of pages. Implementations may restrict the size and alignment of range lockings to be on page-size boundaries. The page size, in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

Memory locking guarantees the residence of portions of the address space. It is implementation-defined whether locking memory guarantees fixed translation between virtual addresses (as seen by the process) and physical addresses. Per-process memory locks are not inherited across a *fork*( ), and all memory locks owned by a process are unlocked upon *exec* or process termination. Unmapping of an address range removes any memory locks established on that address range by this process.

2.8.3.2 Memory Mapped Files

MF The functionality described in this section is dependent on support of the Memory Mapped Files option (and the rest of this section is not further shaded for this option).

Range memory mapping operations are defined in terms of pages. Implementations may restrict the size and alignment of range mappings to be on page-size boundaries. The page size, in bytes, is the value of the configurable system variable {PAGESIZE}. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

Memory mapped files provide a mechanism that allows a process to access files by directly incorporating file data into its address space. Once a file is mapped into a process address space, the data can be manipulated as memory. If more than one process maps a file, its contents are shared among them. If the mappings allow shared write access, then data written into the memory object through the address space of one process appears in the address spaces of all processes that similarly map the same portion of the memory object.

SHM Shared memory objects are named regions of storage that may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

SHM An *unlink*( ) of a file or *shm_unlink*( ) of a shared memory object, while causing the removal of the name, does not unmap any mappings established for the object. Once the name has been removed, the contents of the memory object are preserved as long as it is referenced. The memory object remains referenced as long as a process has the memory object open or has some area of the memory object mapped.

2.8.3.3 Memory Protection

MPR MF The functionality described in this section is dependent on support of the Memory Protection and Memory Mapped Files option (and the rest of this section is not further shaded for these options).

When an object is mapped, various application accesses to the mapped region may result in signals. In this context, SIGBUS is used to indicate an error using the mapped object, and SIGSEGV is used to indicate a protection violation or misuse of an address:

- A mapping may be restricted to disallow some types of access.
- Write attempts to memory that was mapped without write access, or any access to memory mapped PROT_NONE, shall result in a SIGSEGV signal.
- References to unmapped addresses shall result in a SIGSEGV signal.
- Reference to whole pages within the mapping, but beyond the current length of the object, shall result in a SIGBUS signal.
- The size of the object is unaffected by access beyond the end of the object (even if a SIGBUS is not generated).

2.8.3.4 Typed Memory Objects

TYM The functionality described in this section is dependent on support of the Typed Memory Objects option (and the rest of this section is not further shaded for this option).

Implementations may support the Typed Memory Objects option without supporting the Memory Mapped Files option or the Shared Memory Objects option. Typed memory objects are implementation-configurable named storage pools accessible from one or more processors in a system, each via one or more ports, such as backplane buses, LANs, I/O channels, and so on. Each valid combination of a storage pool and a port is identified through a name that is defined at system configuration time, in an implementation-defined manner; the name may be independent of the file system. Using this name, a typed memory object can be opened and mapped into process address space. For a given storage pool and port, it is necessary to support both dynamic allocation from the pool as well as mapping at an application-supplied offset within the pool; when dynamic allocation has been performed, subsequent deallocation must be supported. Lastly, accessing typed memory objects from different ports requires a method for obtaining the offset and length of contiguous storage of a region of typed memory (dynamically allocated or not); this allows typed memory to be shared among processes and/or processors while being accessed from the desired port.

2.8.4 Process Scheduling

PS The functionality described in this section is dependent on support of the Process Scheduling option (and the rest of this section is not further shaded for this option).

Scheduling Policies

The scheduling semantics described in this volume of IEEE Std 1003.1-200x are defined in terms of a conceptual model that contains a set of thread lists. No implementation structures are necessarily implied by the use of this conceptual model. It is assumed that no time elapses during operations described using this model, and therefore no simultaneous operations are possible. This model discusses only processor scheduling for runnable threads, but it should be noted that greatly enhanced predictability of realtime applications results if the sequencing of other resources takes processor scheduling policy into account.

There is, conceptually, one thread list for each priority. A runnable thread will be on the thread list for that thread's priority. Multiple scheduling policies shall be provided. Each non-empty thread list is ordered, contains a head as one end of its order, and a tail as the other. The purpose of a scheduling policy is to define the allowable operations on this set of lists (for example, moving threads between and within lists).

Each process shall be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the *sched_setscheduler*( ) or *sched_setparam*( ) functions.

Each thread shall be controlled by an associated scheduling policy and priority. These parameters may be specified by explicit application execution of the *pthread_setschedparam*( ) function.

Associated with each policy is a priority range. Each policy definition shall specify the minimum priority range for that policy. The priority ranges for each policy may but need not overlap the priority ranges of other policies.

A conforming implementation shall select the thread that is defined as being at the head of the highest priority non-empty thread list to become a running thread, regardless of its associated policy. This thread is then removed from its thread list.

Four scheduling policies are specifically required. Other implementation-defined scheduling policies may be defined. The following symbols are defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**:

SCHED_FIFO	First in, first out (FIFO) scheduling policy.
SCHED_RR	Round robin scheduling policy.
SS SCHED_SPORADIC	Sporadic server scheduling policy.
SCHED_OTHER	Another scheduling policy.

The values of these symbols shall be distinct.

SCHED_FIFO

Conforming implementations shall include a scheduling policy called the FIFO scheduling policy.

Threads scheduled under this policy are chosen from a thread list that is ordered by the time its threads have been on the list without being executed; generally, the head of the list is the thread that has been on the list the longest time, and the tail is the thread that has been on the list the shortest time.

Under the SCHED_FIFO policy, the modification of the definitional thread lists is as follows:

1. When a running thread becomes a preempted thread, it becomes the head of the thread list for its priority.
2. When a blocked thread becomes a runnable thread, it becomes the tail of the thread list for its priority.
3. When a running thread calls the *sched_setscheduler*( ) function, the process specified in the function call is modified to the specified policy and the priority specified by the *param* argument.
4. When a running thread calls the *sched_setparam*( ) function, the priority of the process specified in the function call is modified to the priority specified by the *param* argument.

5. When a running thread calls the *pthread_setschedparam*( ) function, the thread specified in the function call is modified to the specified policy and the priority specified by the *param* argument.

6. When a running thread calls the *pthread_setschedprio*( ) function, the thread specified in the function call is modified to the priority specified by the *prio* argument.

7. If a thread whose policy or priority has been modified other than by *pthread_setschedprio*( ) is a running thread or is runnable, it then becomes the tail of the thread list for its new priority.

8. If a thread whose policy or priority has been modified by *pthread_setschedprio*( ) is a running thread or is runnable, the effect on its position in the thread list depends on the direction of the modification, as follows:
    a. If the priority is raised, the thread becomes the tail of the thread list.
    b. If the priority is unchanged, the thread does not change position in the thread list.
    c. If the priority is lowered, the thread becomes the head of the thread list.

9. When a running thread issues the *sched_yield*( ) function, the thread becomes the tail of the thread list for its priority.

10. At no other time is the position of a thread with this scheduling policy within the thread lists affected.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions when SCHED_FIFO is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

SCHED_RR

Conforming implementations shall include a scheduling policy called the *round robin* scheduling policy. This policy shall be identical to the SCHED_FIFO policy with the additional condition that when the implementation detects that a running thread has been executing as a running thread for a time period of the length returned by the *sched_rr_get_interval*( ) function or longer, the thread shall become the tail of its thread list and the head of that thread list shall be removed and made a running thread.

The effect of this policy is to ensure that if there are multiple SCHED_RR threads at the same priority, one of them does not monopolize the processor. An application should not rely only on the use of SCHED_RR to ensure application progress among multiple threads if the application includes threads using the SCHED_FIFO policy at the same or higher priority levels or SCHED_RR threads at a higher priority level.

A thread under this policy that is preempted and subsequently resumes execution as a running thread completes the unexpired portion of its round robin interval time period.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions when SCHED_RR is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 priorities for this policy.

SCHED_SPORADIC

SS|TSP The functionality described in this section is dependent on support of the Process Sporadic Server or Thread Sporadic Server options (and the rest of this section is not further shaded for these options).

If _POSIX_SPORADIC_SERVER or _POSIX_THREAD_SPORADIC_SERVER is defined, the implementation shall include a scheduling policy identified by the value SCHED_SPORADIC.

The sporadic server policy is based primarily on two parameters: the *replenishment period* and the *available execution capacity*. The replenishment period is given by the *sched_ss_repl_period* member of the **sched_param** structure. The available execution capacity is initialized to the value given by the *sched_ss_init_budget* member of the same parameter. The sporadic server policy is identical to the SCHED_FIFO policy with some additional conditions that cause the thread's assigned priority to be switched between the values specified by the *sched_priority* and *sched_ss_low_priority* members of the **sched_param** structure.

The priority assigned to a thread using the sporadic server scheduling policy is determined in the following manner: if the available execution capacity is greater than zero and the number of pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned the priority specified by *sched_priority*; otherwise, the assigned priority shall be *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the thread list corresponding to its assigned priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned priority, is done as follows:

1. When the thread at the head of the *sched_priority* list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation-defined.

2. Each time the thread is inserted at the tail of the list associated with *sched_priority*—because as a blocked thread it became runnable with priority *sched_priority* or because a replenishment operation was performed—the time at which this operation is done is posted as the *activation_time*.

3. When the running thread with assigned priority equal to *sched_priority* becomes a preempted thread, it becomes the head of the thread list for its priority, and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.

4. When the running thread with assigned priority equal to *sched_priority* becomes a blocked thread, the execution time consumed is subtracted from the available execution capacity, and a replenishment operation is scheduled, as described in 6 and 7. If the available execution capacity would become negative by this operation, it shall be set to zero.

5. When the running thread with assigned priority equal to *sched_priority* reaches the limit imposed on its execution time, it becomes the tail of the thread list for *sched_ss_low_priority*, the execution time consumed is subtracted from the available execution capacity (which becomes zero), and a replenishment operation is scheduled, as described in 6 and 7.

6. Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, *replenish_amount*, is set equal to the execution time consumed by the thread since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the

replenishment operation is carried out immediately. Several replenishment operations may be pending at the same time, each of which will be serviced at its respective scheduled time. With the above rules, the number of replenishment operations simultaneously pending for a given thread that is scheduled under the sporadic server policy shall not be greater than *sched_ss_max_repl*.

7. A replenishment operation consists of adding the corresponding *replenish_amount* to the available execution capacity at the scheduled time. If, as a consequence of this operation, the execution capacity would become larger than *sched_ss_initial_budget*, it shall be rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was runnable or running, and had assigned priority equal to *sched_ss_low_priority*, then it becomes the tail of the thread list for *sched_priority*.

Execution time is defined in Section 2.2.2 (on page 464).

For this policy, changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on its behavior.

For this policy, valid priorities shall be within the range returned by the *sched_get_priority_min*( ) and *sched_get_priority_max*( ) functions when SCHED_SPORADIC is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 distinct priorities for this policy.

SCHED_OTHER

Conforming implementations shall include one scheduling policy identified as SCHED_OTHER (which may execute identically with either the FIFO or round robin scheduling policy). The effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads SS are executing under SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is implementation-defined.

This policy is defined to allow strictly conforming applications to be able to indicate in a portable manner that they no longer need a realtime scheduling policy.

For threads executing under this policy, the implementation shall use only priorities within the range returned by the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions when SCHED_OTHER is provided as the parameter.

2.8.5 Clocks and Timers

TMR The functionality described in this section is dependent on support of the Timers option (and the rest of this section is not further shaded for this option).

The **<time.h>** header defines the types and manifest constants used by the timing facility.

Time Value Specification Structures

Many of the timing facility functions accept or return time value specifications. A time value structure **timespec** specifies a single time value and includes at least the following members:

Member Type	Member Name	Description
time_t	*tv_sec*	Seconds.
long	*tv_nsec*	Nanoseconds.

The *tv_nsec* member is only valid if greater than or equal to zero, and less than the number of nanoseconds in a second (1,000 million). The time interval described by this structure is ($tv\_sec * 10^9 + tv\_nsec$) nanoseconds.

A time value structure **itimerspec** specifies an initial timer value and a repetition interval for use by the per-process timer functions. This structure includes at least the following members:

Member Type	Member Name	Description
struct timespec	*it_interval*	Timer period.
struct timespec	*it_value*	Timer expiration.

If the value described by *it_value* is non-zero, it indicates the time to or time of the next timer expiration (for relative and absolute timer values, respectively). If the value described by *it_value* is zero, the timer shall be disarmed.

If the value described by *it_interval* is non-zero, it specifies an interval which shall be used in reloading the timer when it expires; that is, a periodic timer is specified. If the value described by *it_interval* is zero, the timer is disarmed after its next expiration; that is, a one-shot timer is specified. |

Timer Event Notification Control Block

RTS Per-process timers may be created that notify the process of timer expirations by queuing a realtime extended signal. The **sigevent** structure, defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, is used in creating such a timer. The **sigevent** structure contains the signal number and an application-specific data value which shall be used when notifying the calling process of timer expiration events. |

Manifest Constants

The following constants are defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**:

CLOCK_REALTIME	The identifier for the system-wide realtime clock.
TIMER_ABSTIME	Flag indicating time is absolute with respect to the clock associated with a timer.
MON CLOCK_MONOTONIC	The identifier for the system-wide monotonic clock, which is defined as a clock whose value cannot be set via *clock_settime*( ) and which cannot have backward clock jumps. The maximum possible clock jump is implementation-defined.

MON The maximum allowable resolution for CLOCK_REALTIME and CLOCK_MONOTONIC clocks and all time services based on these clocks is represented by {_POSIX_CLOCKRES_MIN} and shall be defined as 20ms (1/50 of a second). Implementations may support smaller values of resolution for these clocks to provide finer granularity time bases. The actual resolution supported by an implementation for a specific clock is obtained using the *clock_getres*( ) function. If the actual resolution supported for a time service based on one of these clocks differs from the resolution supported for that clock, the implementation shall document this difference.

MON The minimum allowable maximum value for CLOCK_REALTIME and CLOCK_MONOTONIC clocks and all absolute time services based on them is the same as that defined by the ISO C standard for the **time_t** type. If the maximum value supported by a time service based on one of these clocks differs from the maximum value supported by that clock, the implementation shall document this difference.

Execution Time Monitoring

CPT If _POSIX_CPUTIME is defined, process CPU-time clocks shall be supported in addition to the clocks described in **Manifest Constants** (on page 499).

TCT If _POSIX_THREAD_CPUTIME is defined, thread CPU-time clocks shall be supported.

CPT|TCT CPU-time clocks measure execution or CPU time, which is defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.117, CPU Time (Execution Time). The mechanism used to measure execution time is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.9, Measurement of Execution Time. |

CPT If _POSIX_CPUTIME is defined, the following constant of the type **clockid_t** is defined in **<time.h>**: |

CLOCK_PROCESS_CPUTIME_ID
When this value of the type **clockid_t** is used in a *clock*( ) or *timer**( ) function call, it is interpreted as the identifier of the CPU-time clock associated with the process making the function call.

TCT If _POSIX_THREAD_CPUTIME is defined, the following constant of the type **clockid_t** is defined in **<time.h>**: |

CLOCK_THREAD_CPUTIME_ID
When this value of the type **clockid_t** is used in a *clock*( ) or *timer**( ) function call, it is interpreted as the identifier of the CPU-time clock associated with the thread making the function call.

2.9 Threads

THR The functionality described in this section is dependent on support of the Threads option (and the rest of this section is not further shaded for this option).

This section defines functionality to support multiple flows of control, called *threads*, within a process. For the definition of *threads*, see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.393, Thread.

The specific functional areas covered by threads and their scope include:

- Thread management: the creation, control, and termination of multiple flows of control in the same process under the assumption of a common shared address space
- Synchronization primitives optimized for tightly coupled operation of multiple control flows in a common, shared address space

2.9.1 Thread-Safety

All functions defined by this volume of IEEE Std 1003.1-200x shall be thread-safe, except that the following functions[1] need not be thread-safe.

---

[1] The functions in the table are not shaded to denote applicable options. Individual reference pages should be consulted.

asctime( )
basename( )
catgets( )
crypt( )
ctime( )
dbm_clearerr( )
dbm_close( )
dbm_delete( )
dbm_error( )
dbm_fetch( )
dbm_firstkey( )
dbm_nextkey( )
dbm_open( )
dbm_store( )
dirname( )
dlerror( )
drand48( )
ecvt( )
encrypt( )
endgrent( )
endpwent( )
endutxent( )
fcvt( )
ftw( )
gcvt( )
getc_unlocked( )
getchar_unlocked( )
getdate( )
getenv( )
getgrent( )
getgrgid( )
getgrnam( )
gethostbyaddr( )
gethostbyname( )
gethostent( )
getlogin( )
getnetbyaddr( )
getnetbyname( )
getnetent( )
getopt( )
getprotobyname( )
getprotobynumber( )
getprotoent( )
getpwent( )
getpwnam( )
getpwuid( )
getservbyname( )
getservbyport( )
getservent( )
getutxent( )
getutxid( )
getutxline( )
gmtime( )
hcreate( )
hdestroy( )
hsearch( )
inet_ntoa( )
l64a( )
lgamma( )
localeconv( )
localtime( )
lrand48( )
mrand48( )
nftw( )
nl_langinfo( )
ptsname( )
putc_unlocked( )
putchar_unlocked( )
putenv( )
pututxline( )
rand( )
readdir( )
setenv( )
setgrent( )
setkey( )
setpwent( )
setutxent( )
strerror( )
strtok( )
ttyname( )
unsetenv( )
wcstombs( )
wctomb( )

The *ctermid*( ) and *tmpnam*( ) functions need not be thread-safe if passed a NULL argument. The *wcrtomb*( ) and *wcsrtombs*( ) functions need not be thread-safe if passed a NULL *ps* argument.

Implementations shall provide internal synchronization as necessary in order to satisfy this requirement.

2.9.2 Thread IDs

Although implementations may have thread IDs that are unique in a system, applications should only assume that thread IDs are usable and unique within a single process. The effect of calling any of the functions defined in this volume of IEEE Std 1003.1-200x and passing as an argument the thread ID of a thread from another process is unspecified. A conforming implementation is free to reuse a thread ID after the thread terminates if it was created with the *detachstate* attribute set to PTHREAD_CREATE_DETACHED or if *pthread_detach*( ) or *pthread_join*( ) has been called for that thread. If a thread is detached, its thread ID is invalid for use as an argument in a call to *pthread_detach*( ) or *pthread_join*( ).

2.9.3 Thread Mutexes

A thread that has blocked shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources is determined by the scheduling policy.

A thread shall become the owner of a mutex, *m*, when one of the following occurs:

- It returns successfully from *pthread_mutex_lock*( ) with *m* as the *mutex* argument.
- It returns successfully from *pthread_mutex_trylock*( ) with *m* as the *mutex* argument.
- TMO It returns successfully from *pthread_mutex_timedwait*( ) with *m* as the mutex argument.
- It returns (successfully or not) from *pthread_cond_wait*( ) with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors).
- It returns (successfully or not) from *pthread_cond_timedwait*( ) with *m* as the *mutex* argument (except as explicitly indicated otherwise for certain errors).

The thread shall remain the owner of *m* until one of the following occurs:

- It executes *pthread_mutex_unlock*( ) with *m* as the *mutex* argument
- It blocks in a call to *pthread_cond_wait*( ) with *m* as the *mutex* argument.
- It blocks in a call to *pthread_cond_timedwait*( ) with *m* as the *mutex* argument.

The implementation shall behave as if at all times there is at most one owner of any mutex.

A thread that becomes the owner of a mutex is said to have *acquired* the mutex and the mutex is said to have become *locked*; when a thread gives up ownership of a mutex it is said to have *released* the mutex and the mutex is said to have become *unlocked*.

2.9.4 Thread Scheduling

TPS The functionality described in this section is dependent on support of the Thread Execution Scheduling option (and the rest of this section is not further shaded for this option).

Thread Scheduling Attributes

In support of the scheduling function, threads have attributes which are accessed through the **pthread_attr_t** thread creation attributes object.

The *contentionscope* attribute defines the scheduling contention scope of the thread to be either PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM.

The *inheritsched* attribute specifies whether a newly created thread is to inherit the scheduling attributes of the creating thread or to have its scheduling values set according to the other scheduling attributes in the **pthread_attr_t** object.

The *schedpolicy* attribute defines the scheduling policy for the thread. The *schedparam* attribute defines the scheduling parameters for the thread. The interaction of threads having different policies within a process is described as part of the definition of those policies.

If the Thread Execution Scheduling option is defined, and the *schedpolicy* attribute specifies one of the priority-based policies defined under this option, the *schedparam* attribute contains the scheduling priority of the thread. A conforming implementation ensures that the priority value in *schedparam* is in the range associated with the scheduling policy when the thread attributes object is used to create a thread, or when the scheduling attributes of a thread are dynamically modified. The meaning of the priority value in *schedparam* is the same as that of *priority*.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the *schedparam* attribute supports four new members that are used for the sporadic server scheduling policy. These members are *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The meaning of these attributes is the same as in the definitions that appear under Section 2.8.4 (on page 494).

When a process is created, its single thread has a scheduling policy and associated attributes equal to the process' policy and attributes. The default scheduling contention scope value is implementation-defined. The default values of other scheduling attributes are implementation-defined.

Thread Scheduling Contention Scope

The scheduling contention scope of a thread defines the set of threads with which the thread competes for use of the processing resources. The scheduling operation selects at most one thread to execute on each processor at any point in time and the thread's scheduling attributes (for example, *priority*), whether under process scheduling contention scope or system scheduling contention scope, are the parameters used to determine the scheduling decision.

The scheduling contention scope, in the context of scheduling a mixed scope environment, affects threads as follows:

- A thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope contends for resources with all other threads in the same scheduling allocation domain relative to their system scheduling attributes. The system scheduling attributes of a thread created with PTHREAD_SCOPE_SYSTEM scheduling contention scope are the scheduling attributes with which the thread was created. The system scheduling attributes of a thread created with PTHREAD_SCOPE_PROCESS scheduling contention scope are the implementation-defined mapping into system attribute space of the scheduling attributes with which the thread was created.
- Threads created with PTHREAD_SCOPE_PROCESS scheduling contention scope contend directly with other threads within their process that were created with PTHREAD_SCOPE_PROCESS scheduling contention scope. The contention is resolved based on the threads' scheduling attributes and policies. It is unspecified how such threads are scheduled relative to threads in other processes or threads with PTHREAD_SCOPE_SYSTEM scheduling contention scope.
- Conforming implementations shall support the PTHREAD_SCOPE_PROCESS scheduling contention scope, the PTHREAD_SCOPE_SYSTEM scheduling contention scope, or both.

Scheduling Allocation Domain

Implementations shall support scheduling allocation domains containing one or more processors. It should be noted that the presence of multiple processors does not automatically indicate a scheduling allocation domain size greater than one. Conforming implementations on multi-processors may map all or any subset of the CPUs to one or multiple scheduling allocation domains, and could define these scheduling allocation domains on a per-thread, per-process, or per-system basis, depending on the types of applications intended to be supported by the implementation. The scheduling allocation domain is independent of scheduling contention scope, as the scheduling contention scope merely defines the set of threads with which a thread contends for processor resources, while scheduling allocation domain defines the set of processors for which it contends. The semantics of how this contention is resolved among threads for processors is determined by the scheduling policies of the threads.

The choice of scheduling allocation domain size and the level of application control over scheduling allocation domains is implementation-defined. Conforming implementations may change the size of scheduling allocation domains and the binding of threads to scheduling allocation domains at any time.

For application threads with scheduling allocation domains of size equal to one, the scheduling rules defined for SCHED_FIFO and SCHED_RR shall be used; see **Scheduling Policies** (on page 494). All threads with system scheduling contention scope, regardless of the processes in which they reside, compete for the processor according to their priorities. Threads with process scheduling contention scope compete only with other threads with process scheduling contention scope within their process.

TSP For application threads with scheduling allocation domains of size greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC shall be used in an implementation-defined manner. Each thread with system scheduling contention scope competes for the processors in its scheduling allocation domain in an implementation-defined manner according to its priority. Threads with process scheduling contention scope are scheduled relative to other threads within the same scheduling contention scope in the process.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, the rules defined for SCHED_SPORADIC in **Scheduling Policies** (on page 494) shall be used in an implementation-defined manner for application threads whose scheduling allocation domain size is greater than one.

Scheduling Documentation

TSP If _POSIX_PRIORITY_SCHEDULING is defined, then any scheduling policies beyond SCHED_OTHER, SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC, as well as the effects of the scheduling policies indicated by these other values, and the attributes required in order to support such a policy, are implementation-defined. Furthermore, the implementation shall document the effect of all processor scheduling allocation domain values supported for these policies.

2.9.5 Thread Cancelation

The thread cancelation mechanism allows a thread to terminate the execution of any other thread in the process in a controlled manner. The target thread (that is, the one that is being canceled) is allowed to hold cancelation requests pending in a number of ways and to perform application-specific cleanup processing when the notice of cancelation is acted upon.

Cancelation is controlled by the cancelation control functions. Each thread maintains its own cancelability state. Cancelation may only occur at cancelation points or when the thread is asynchronously cancelable.

The thread cancelation mechanism described in this section depends upon programs having set *deferred cancelability* state, which is specified as the default. Applications shall also carefully follow static lexical scoping rules in their execution behavior. For example, use of *setjmp*( ), *return*, *goto*, and so on, to leave user-defined cancelation scopes without doing the necessary scope pop operation results in undefined behavior.

Use of asynchronous cancelability while holding resources which potentially need to be released may result in resource loss. Similarly, cancelation scopes may only be safely manipulated (pushed and popped) when the thread is in the *deferred* or *disabled* cancelability states.

2.9.5.1 Cancelability States

The cancelability state of a thread determines the action taken upon receipt of a cancelation request. The thread may control cancelation in a number of ways.

Each thread maintains its own cancelability state, which may be encoded in two bits:

1. Cancelability-Enable: When cancelability is PTHREAD_CANCEL_DISABLE (as defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**), cancelation requests against the target thread are held pending. By default, cancelability is set to PTHREAD_CANCEL_ENABLE (as defined in **<pthread.h>**).

2. Cancelability Type: When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_ASYNCHRONOUS (as defined in **<pthread.h>**), new or pending cancelation requests may be acted upon at any time. When cancelability is enabled and the cancelability type is PTHREAD_CANCEL_DEFERRED (as defined in **<pthread.h>**),

cancelation requests are held pending until a cancelation point (see below) is reached. If cancelability is disabled, the setting of the cancelability type has no immediate effect as all cancelation requests are held pending; however, once cancelability is enabled again the new type is in effect. The cancelability type is PTHREAD_CANCEL_DEFERRED in all newly created threads including the thread in which *main*( ) was first invoked.

2.9.5.2 Cancelation Points

Cancelation points shall occur when a thread is executing the following functions:

accept( )
aio_suspend( )
clock_nanosleep( )
close( )
connect( )
creat( )
fcntl( )[2]
fsync( )
getmsg( )
getpmsg( )
lockf( )
mq_receive( )
mq_send( )
mq_timedreceive( )
mq_timedsend( )
msgrcv( )
msgsnd( )
msync( )
nanosleep( )
open( )
pause( )
poll( )
pread( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_join( )
pthread_testcancel( )
putmsg( )
putpmsg( )
pwrite( )
read( )
readv( )
recv( )
recvfrom( )
recvmsg( )
select( )
sem_timedwait( )
sem_wait( )
send( )
sendmsg( )
sendto( )
sigpause( )
sigsuspend( )
sigtimedwait( )
sigwait( )
sigwaitinfo( )
sleep( )
system( )
tcdrain( )
usleep( )
wait( )
waitid( )
waitpid( )
write( )
writev( )

---

[2] When the *cmd* argument is F_SETLKW.

A cancelation point may also occur when a thread is executing the following functions:

catclose( )
catgets( )
catopen( )
closedir( )
closelog( )
ctermid( )
dbm_close( )
dbm_delete( )
dbm_fetch( )
dbm_nextkey( )
dbm_open( )
dbm_store( )
dlclose( )
dlopen( )
endgrent( )
endhostent( )
endnetent( )
endprotoent( )
endpwent( )
endservent( )
endutxent( )
fclose( )
fcntl( )[3]
fflush( )
fgetc( )
fgetpos( )
fgets( )
fgetwc( )
fgetws( )
fopen( )
fprintf( )
fputc( )
fputs( )
fputwc( )
fputws( )
fread( )
freopen( )
fscanf( )
fseek( )
fseeko( )
fsetpos( )
ftell( )
ftello( )
ftw( )
fwprintf( )
fwrite( )
fwscanf( )
getc( )
getc_unlocked( )
getchar( )
getchar_unlocked( )
getcwd( )
getdate( )
getgrent( )
getgrgid( )
getgrgid_r( )
getgrnam( )
getgrnam_r( )
gethostbyaddr( )
gethostbyname( )
gethostent( )
gethostname( )
getlogin( )
getlogin_r( )
getnetbyaddr( )
getnetbyname( )
getnetent( )
getprotobyname( )
getprotobynumber( )
getprotoent( )
getpwent( )
getpwnam( )
getpwnam_r( )
getpwuid( )
getpwuid_r( )
gets( )
getservbyname( )
getservbyport( )
getservent( )
getutxent( )
getutxid( )
getutxline( )
getwc( )
getwchar( )
getwd( )
glob( )
iconv_close( )
iconv_open( )
ioctl( )
lseek( )
mkstemp( )
nftw( )
opendir( )
openlog( )
pclose( )
perror( )
popen( )
posix_fadvise( )
posix_fallocate( )
posix_madvise( )
posix_spawn( )
posix_spawnp( )
posix_trace_clear( )
posix_trace_close( )
posix_trace_create( )
posix_trace_create_withlog( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_flush( )
posix_trace_get_attr( )
posix_trace_get_filter( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_open( )
posix_trace_rewind( )
posix_trace_set_filter( )
posix_trace_shutdown( )
posix_trace_timedgetnext_event( )
posix_typed_mem_open( )
printf( )
pthread_rwlock_rdlock( )
pthread_rwlock_timedrdlock( )
pthread_rwlock_timedwrlock( )
pthread_rwlock_wrlock( )
putc( )
putc_unlocked( )
putchar( )
putchar_unlocked( )
puts( )
pututxline( )
putwc( )
putwchar( )
readdir( )
readdir_r( )
remove( )
rename( )
rewind( )
rewinddir( )
scanf( )
seekdir( )
semop( )
setgrent( )
sethostent( )
setnetent( )
setprotoent( )
setpwent( )
setservent( )
setutxent( )
strerror( )
syslog( )
tmpfile( )
tmpnam( )
ttyname( )
ttyname_r( )
ungetc( )
ungetwc( )
unlink( )
vfprintf( )
vfwprintf( )
vprintf( )
vwprintf( )
wprintf( )
wscanf( )

An implementation shall not introduce cancelation points into any other functions specified in this volume of IEEE Std 1003.1-200x.

---

[3] For any value of the *cmd* argument.

The side effects of acting upon a cancelation request while suspended during a call of a function are the same as the side effects that may be seen in a single-threaded program when a call to a function is interrupted by a signal and the given function returns [EINTR]. Any such side effects occur before any cancelation cleanup handlers are called.

Whenever a thread has cancelability enabled and a cancelation request has been made with that thread as the target, and the thread then calls any function that is a cancelation point (such as *pthread_testcancel*( ) or *read*( )), the cancelation request shall be acted upon before the function returns. If a thread has cancelability enabled and a cancelation request is made with the thread as a target while the thread is suspended at a cancelation point, the thread shall be awakened and the cancelation request shall be acted upon. However, if the thread is suspended at a cancelation point and the event for which it is waiting occurs before the cancelation request is acted upon, it is unspecified whether the cancelation request is acted upon or whether the cancelation request remains pending and the thread resumes normal execution.

2.9.5.3 *Thread Cancelation Cleanup Handlers*

Each thread maintains a list of cancelation cleanup handlers. The programmer uses the *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions to place routines on and remove routines from this list.

When a cancelation request is acted upon, the routines in the list are invoked one by one in LIFO sequence; that is, the last routine pushed onto the list (Last In) is the first to be invoked (First Out). The thread invokes the cancelation cleanup handler with cancelation disabled until the last cancelation cleanup handler returns. When the cancelation cleanup handler for a scope is invoked, the storage for that scope remains valid. If the last cancelation cleanup handler returns, thread execution is terminated and a status of PTHREAD_CANCELED is made available to any threads joining with the target. The symbolic constant PTHREAD_CANCELED expands to a constant expression of type (**void ***) whose value matches no pointer to an object in memory nor the value NULL.

The cancelation cleanup handlers are also invoked when the thread calls *pthread_exit*( ).

A side effect of acting upon a cancelation request while in a condition variable wait is that the mutex is re-acquired before calling the first cancelation cleanup handler. In addition, the thread is no longer considered to be waiting for the condition and the thread shall not have consumed any pending condition signals on the condition.

A cancelation cleanup handler cannot exit via *longjmp*( ) or *siglongjmp*( ).

2.9.5.4 *Async-Cancel Safety*

The *pthread_cancel*( ), *pthread_setcancelstate*( ), and *pthread_setcanceltype*( ) functions are defined to be async-cancel safe.

No other functions in this volume of IEEE Std 1003.1-200x are required to be async-cancel-safe.

2.9.6 Thread Read-Write Locks

Multiple readers, single writer (read-write) locks allow many threads to have simultaneous read-only access to data while allowing only one thread to have exclusive write access at any given time. They are typically used to protect data that is read more frequently than it is changed.

One or more readers acquire read access to the resource by performing a read lock operation on the associated read-write lock. A writer acquires exclusive write access by performing a write lock operation. Basically, all readers exclude any writers and a writer excludes all readers and any other writers.

A thread that has blocked on a read-write lock (for example, has not yet returned from a *pthread_rwlock_rdlock*( ) or *pthread_rwlock_wrlock*( ) call) shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources shall be determined by the scheduling policy.

Read-write locks can be used to synchronize threads in the current process and other processes if they are allocated in memory that is writable and shared among the cooperating processes and have been initialized for this behavior.

2.9.7 Thread Interactions with Regular File Operations

All of the functions *chmod*( ), *close*( ), *fchmod*( ), *fcntl*( ), *fstat*( ), *ftruncate*( ), *lseek*( ), *open*( ), *read*( ), *readlink*( ), *stat*( ), *symlink*( ), and *write*( ) shall be atomic with respect to each other in the effects specified in IEEE Std 1003.1-200x when they operate on regular files. If two threads each call one of these functions, each call shall either see all of the specified effects of the other call, or none of them.

2.10 Sockets

A socket is an endpoint for communication using the facilities described in this section. A socket is created with a specific socket type, described in Section 2.10.6 (on page 509), and is associated with a specific protocol, detailed in Section 2.10.3 (on page 509). A socket is accessed via a file descriptor obtained when the socket is created.

2.10.1 Address Families

All network protocols are associated with a specific address family. An address family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. An address family is normally comprised of a number of protocols, one per socket type. Each protocol is characterized by an abstract socket type. It is not required that an address family support all socket types. An address family may contain multiple protocols supporting the same socket abstraction.

Section 2.10.17 (on page 516), Section 2.10.19 (on page 517), and Section 2.10.20 (on page 517), respectively, describe the use of sockets for local UNIX connections, for Internet protocols based on IPv4, and for Internet protocols based on IPv6.

2.10.2 Addressing

An address family defines the format of a socket address. All network addresses are described using a general structure, called a **sockaddr**, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**. However, each address family imposes finer and more specific structure, generally defining a structure with fields specific to the address family. The field *sa_family* in the **sockaddr** structure contains the address family identifier, specifying the format of the *sa_data* area. The size of the *sa_data* area is unspecified.

2.10.3 Protocols

A protocol supports one of the socket abstractions detailed in Section 2.10.6. Selecting a protocol involves specifying the address family, socket type, and protocol number to the *socket*( ) function. Certain semantics of the basic socket abstractions are protocol-specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism.

2.10.4 Routing

Sockets provides packet routing facilities. A routing information database is maintained, which is used in selecting the appropriate network interface when transmitting packets.

2.10.5 Interfaces

Each network interface in a system corresponds to a path through which messages can be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, do not.

2.10.6 Socket Types

A socket is created with a specific type, which defines the communication semantics and which allows the selection of an appropriate communication protocol. Four types are defined: RS SOCK_RAW, SOCK_STREAM, SOCK_SEQPACKET, and SOCK_DGRAM. Implementations may specify additional socket types.

The SOCK_STREAM socket type provides reliable, sequenced, full-duplex octet streams between the socket and a peer to which the socket is connected. A socket of type SOCK_STREAM must be in a connected state before any data may be sent or received. Record boundaries are not maintained; data sent on a stream socket using output operations of one size may be received using input operations of smaller or larger sizes without loss of data. Data may be buffered; successful return from an output function does not imply that the data has been delivered to the peer or even transmitted from the local system. If data cannot be successfully transmitted within a given time then the connection is considered broken, and subsequent operations shall fail. A SIGPIPE signal is raised if a thread sends on a broken stream (one that is no longer connected). Support for an out-of-band data transmission facility is protocol-specific.

The SOCK_SEQPACKET socket type is similar to the SOCK_STREAM type, and is also connection-oriented. The only difference between these types is that record boundaries are maintained using the SOCK_SEQPACKET type. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers parts of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag in the received message flags returned by the *recvmsg*( ) function. It is protocol-specific whether a maximum record size is imposed.

The SOCK_DGRAM socket type supports connectionless data transfer which is not necessarily acknowledged or reliable. Datagrams may be sent to the address specified (possibly multicast or broadcast) in each output operation, and incoming datagrams may be received from multiple sources. The source address of each datagram is available when receiving the datagram. An application may also pre-specify a peer address, in which case calls to output functions shall send to the pre-specified peer. If a peer has been specified, only datagrams from that peer shall be received. A datagram must be sent in a single output operation, and must be received in a single input operation. The maximum size of a datagram is protocol-specific; with some protocols, the limit is implementation-defined. Output datagrams may be buffered within the system; thus, a successful return from an output function does not guarantee that a datagram is actually sent or received. However, implementations should attempt to detect any errors possible before the return of an output function, reporting any error by an unsuccessful return value.

RS The SOCK_RAW socket type is similar to the SOCK_DGRAM type. It differs in that it is normally used with communication providers that underlie those used for the other socket types. For this reason, the creation of a socket with type SOCK_RAW shall require appropriate privilege. The format of datagrams sent and received with this socket type generally include specific protocol headers, and the formats are protocol-specific and implementation-defined.

2.10.7 Socket I/O Mode

The I/O mode of a socket is described by the O_NONBLOCK file status flag which pertains to the open file description for the socket. This flag is initially off when a socket is created, but may be set and cleared by the use of the F_SETFL command of the *fcntl*( ) function.

When the O_NONBLOCK flag is set, functions that would normally block until they are complete shall either return immediately with an error, or shall complete asynchronously to the execution of the calling process. Data transfer operations (the *read*( ), *write*( ), *send*( ), and *recv*( ) functions) shall complete immediately, transfer only as much as is available, and then return without blocking, or return an error indicating that no transfer could be made without blocking. The *connect*( ) function initiates a connection and shall return without blocking when O_NONBLOCK is set; it shall return the error [EINPROGRESS] to indicate that the connection was initiated successfully, but that it has not yet completed.

2.10.8 Socket Owner

The owner of a socket is unset when a socket is created. The owner may be set to a process ID or process group ID using the F_SETOWN command of the *fcntl*( ) function.

2.10.9 Socket Queue Limits

The transmit and receive queue sizes for a socket are set when the socket is created. The default sizes used are both protocol-specific and implementation-defined. The sizes may be changed using the *setsockopt*( ) function.

2.10.10 Pending Error

Errors may occur asynchronously, and be reported to the socket in response to input from the network protocol. The socket stores the pending error to be reported to a user of the socket at the next opportunity. The error is returned in response to a subsequent *send*( ), *recv*( ), or *getsockopt*( ) operation on the socket, and the pending error is then cleared.

2.10.11 Socket Receive Queue

A socket has a receive queue that buffers data when it is received by the system until it is removed by a receive call. Depending on the type of the socket and the communication provider, the receive queue may also contain ancillary data such as the addressing and other protocol data associated with the normal data in the queue, and may contain out-of-band or expedited data. The limit on the queue size includes any normal, out-of-band data, datagram source addresses, and ancillary data in the queue. The description in this section applies to all sockets, even though some elements cannot be present in some instances.

The contents of a receive buffer are logically structured as a series of data segments with associated ancillary data and other information. A data segment may contain normal data or out-of-band data, but never both. A data segment may complete a record if the protocol supports records (always true for types SOCK_SEQPACKET and SOCK_DGRAM). A record may be stored as more than one segment; the complete record might never be present in the receive buffer at one time, as a portion might already have been returned to the application, and another portion might not yet have been received from the communications provider. A data segment may contain ancillary protocol data, which is logically associated with the segment. Ancillary data is received as if it were queued along with the first normal data octet in the segment (if any). A segment may contain ancillary data only, with no normal or out-of-band data. For the purposes of this section, a datagram is considered to be a data segment that terminates a record, and that includes a source address as a special type of ancillary data. Data segments are placed into the queue as data is delivered to the socket by the protocol. Normal data segments are placed at the end of the queue as they are delivered. If a new segment contains the same type of data as the preceding segment and includes no ancillary data, and if the preceding segment does not terminate a record, the segments are logically merged into a single segment.

The receive queue is logically terminated if an end-of-file indication has been received or a connection has been terminated. A segment shall be considered to be terminated if another segment follows it in the queue, if the segment completes a record, or if an end-of-file or other connection termination has been reported. The last segment in the receive queue shall also be considered to be terminated while the socket has a pending error to be reported.

A receive operation shall never return data or ancillary data from more than one segment.

2.10.12 Socket Out-of-Band Data State

The handling of received out-of-band data is protocol-specific. Out-of-band data may be placed in the socket receive queue, either at the end of the queue or before all normal data in the queue. In this case, out-of-band data is returned to an application program by a normal receive call. Out-of-band data may also be queued separately rather than being placed in the socket receive queue, in which case it shall be returned only in response to a receive call that requests out-of-band data. It is protocol-specific whether an out-of-band data mark is placed in the receive queue to demarcate data preceding the out-of-band data and following the out-of-band data. An out-of-band data mark is logically an empty data segment that cannot be merged with other segments in the queue. An out-of-band data mark is never returned in response to an input operation. The *sockatmark*( ) function can be used to test whether an out-of-band data mark is the first element in the queue. If an out-of-band data mark is the first element in the queue when an input function is called without the MSG_PEEK option, the mark is removed from the queue and the following data (if any) is processed as if the mark had not been present.

2.10.13 Connection Indication Queue

Sockets that are used to accept incoming connections maintain a queue of outstanding connection indications. This queue is a list of connections that are awaiting acceptance by the application; see *listen*( ).

2.10.14 Signals

One category of event at the socket interface is the generation of signals. These signals report protocol events or process errors relating to the state of the socket. The generation or delivery of a signal does not change the state of the socket, although the generation of the signal may have been caused by a state change.

The SIGPIPE signal shall be sent to a thread that attempts to send data on a socket that is no longer able to send. In addition, the send operation fails with the error [EPIPE].

If a socket has an owner, the SIGURG signal is sent to the owner of the socket when it is notified of expedited or out-of-band data. The socket state at this time is protocol-dependent, and the status of the socket is specified in Section 2.10.17 (on page 516), Section 2.10.19 (on page 517), and Section 2.10.20 (on page 517). Depending on the protocol, the expedited data may or may not have arrived at the time of signal generation.

2.10.15 Asynchronous Errors

If any of the following conditions occur asynchronously for a socket, the corresponding value listed below shall become the pending error for the socket:

[ECONNABORTED]
The connection was aborted locally.

[ECONNREFUSED]
For a connection-mode socket attempting a non-blocking connection, the attempt to connect was forcefully rejected. For a connectionless-mode socket, an attempt to deliver a datagram was forcefully rejected.

[ECONNRESET]
The peer has aborted the connection.

[EHOSTDOWN]
The destination host has been determined to be down or disconnected.

[EHOSTUNREACH]
The destination host is not reachable.

[EMSGSIZE]
For a connectionless-mode socket, the size of a previously sent datagram prevented delivery.

[ENETDOWN]
The local network connection is not operational.

[ENETRESET]
The connection was aborted by the network.

[ENETUNREACH]
The destination network is not reachable.

2.10.16 Use of Options

There are a number of socket options which either specialize the behavior of a socket or provide useful information. These options may be set at different protocol levels and are always present at the uppermost ‘‘socket’’ level.

Socket options are manipulated by two functions, *getsockopt*( ) and *setsockopt*( ). These functions allow an application program to customize the behavior and characteristics of a socket to provide the desired effect.

All of the options have default values. The type and meaning of these values is defined by the protocol level to which they apply. Instead of using the default values, an application program may choose to customize one or more of the options. However, in the bulk of cases, the default values are sufficient for the application.

Some of the options are used to enable or disable certain behavior within the protocol modules (for example, turn on debugging) while others may be used to set protocol-specific information (for example, IP time-to-live on all the application’s outgoing packets). As each of the options is introduced, its effect on the underlying protocol modules is described.

Table 2-1 shows the value for the socket level.

Table 2-1 Value of Level for Socket Options

Name	Description
SOL_SOCKET	Options are intended for the sockets level.

Table 2-2 (on page 514) lists those options present at the socket level; that is, when the *level* parameter of the *getsockopt*( ) or *setsockopt*( ) function is SOL_SOCKET, the types of the option value parameters associated with each option, and a brief synopsis of the meaning of the option value parameter. Unless otherwise noted, each may be examined with *getsockopt*( ) and set with *setsockopt*( ) on all types of socket.

Table 2-2 Socket-Level Options

Option	Parameter Type	Parameter Meaning
SO_BROADCAST	**int**	Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only).
SO_DEBUG	**int**	Non-zero requests debugging in underlying protocol modules.
SO_DONTROUTE	**int**	Non-zero requests bypass of normal routing; route based on destination address only.
SO_ERROR	**int**	Requests and clears pending error information on the socket (*getsockopt*( ) only).
SO_KEEPALIVE	**int**	Non-zero requests periodic transmission of keepalive messages (protocol-specific).
SO_LINGER	**struct linger**	Specify actions to be taken for queued, unsent data on *close*( ): linger on/off and linger time in seconds.
SO_OOBINLINE	**int**	Non-zero requests that out-of-band data be placed into normal data input queue as received.
SO_RCVBUF	**int**	Size of receive buffer (in bytes).
SO_RCVLOWAT	**int**	Minimum amount of data to return to application for input operations (in bytes).
SO_RCVTIMEO	**struct timeval**	Timeout value for a socket receive operation.
SO_REUSEADDR	**int**	Non-zero requests reuse of local addresses in *bind*( ) (protocol-specific).
SO_SNDBUF	**int**	Size of send buffer (in bytes).
SO_SNDLOWAT	**int**	Minimum amount of data to send for output operations (in bytes).
SO_SNDTIMEO	**struct timeval**	Timeout value for a socket send operation.
SO_TYPE	**int**	Identify socket type (*getsockopt*( ) only).

The SO_BROADCAST option requests permission to send broadcast datagrams on the socket. Support for SO_BROADCAST is protocol-specific. The default for SO_BROADCAST is that the ability to send broadcast datagrams on a socket is disabled.

The SO_DEBUG option enables debugging in the underlying protocol modules. This can be useful for tracing the behavior of the underlying protocol modules during normal system operation. The semantics of the debug reports are implementation-defined. The default value for SO_DEBUG is for debugging to be turned off.

The SO_DONTROUTE option requests that outgoing messages bypass the standard routing facilities. The destination must be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. It is protocol-specific whether this option has any effect and how the outgoing network interface is chosen. Support for this option with each protocol is implementation-defined.

The SO_ERROR option is used only on *getsockopt*( ). When this option is specified, *getsockopt*( ) shall return any pending error on the socket and clear the error status. It shall return a value of 0 if there is no pending error. SO_ERROR may be used to check for asynchronous errors on connected connectionless-mode sockets or for other types of asynchronous errors. SO_ERROR has no default value.

The SO_KEEPALIVE option enables the periodic transmission of messages on a connected socket. The behavior of this option is protocol-specific. The default value for SO_KEEPALIVE is zero, specifying that this capability is turned off.

The SO_LINGER option controls the action of the interface when unsent messages are queued on a socket and a *close*( ) is performed. The details of this option are protocol-specific. The default value for SO_LINGER is zero, or off, for the *l_onoff* element of the option value and zero seconds for the linger time specified by the *l_linger* element.

The SO_OOBINLINE option is valid only on protocols that support out-of-band data. The SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it is then accessible using the *read*( ) or *recv*( ) functions without the MSG_OOB flag set. The default for SO_OOBINLINE is off; that is, for out-of-band data not to be placed in the normal data input queue.

The SO_RCVBUF option requests that the buffer space allocated for receive operations on this socket be set to the value, in bytes, of the option value. Applications may wish to increase buffer size for high volume connections, or may decrease buffer size to limit the possible backlog of incoming data. The default value for the SO_RCVBUF option value is implementation-defined, and may vary by protocol.

The maximum value for the option for a socket may be obtained by the use of the *fpathconf*( ) function, using the value _PC_SOCK_MAXBUF.

The SO_RCVLOWAT option sets the minimum number of bytes to process for socket input operations. In general, receive calls block until any (non-zero) amount of data is received, then return the smaller of the amount available or the amount requested. The default value for SO_RCVLOWAT is 1, and does not affect the general case. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned (for example, out-of-band data). As mentioned previously, the default value for SO_RCVLOWAT is 1 byte. It is implementation-defined whether the SO_RCVLOWAT option can be set.

The SO_RCVTIMEO option is an option to set a timeout value for input operations. It accepts a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* shall be set to [EWOULDBLOCK] if no data were received. The default for this option is the value zero, which indicates that a receive operation will not timeout. It is implementation-defined whether the SO_RCVTIMEO option can be set.

The SO_REUSEADDR option indicates that the rules used in validating addresses supplied in a *bind*( ) should allow reuse of local addresses. Operation of this option is protocol-specific. The default value for SO_REUSEADDR is off; that is, reuse of local addresses is not permitted.

The SO_SNDBUF option requests that the buffer space allocated for send operations on this socket be set to the value, in bytes, of the option value. The default value for the SO_SNDBUF option value is implementation-defined, and may vary by protocol. The maximum value for the option for a socket may be obtained by the use of the *fpathconf*( ) function, using the value _PC_SOCK_MAXBUF.

The SO_SNDLOWAT option sets the minimum number of bytes to process for socket output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Non-blocking output operations process as much data as permitted subject to flow control without blocking, but process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. A *select*( ) operation testing the ability to write to a socket shall return true only if the send low water mark could be processed. The default value for SO_SNDLOWAT is implementation-defined and protocol-specific. It is implementation-defined whether the SO_SNDLOWAT option can be set.

The SO_SNDTIMEO option is an option to set a timeout value for the amount of time that an output function shall block because flow control prevents data from being sent. As noted in Table 2-2 (on page 514), the option value is a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an output operation to complete. If a send operation has blocked for this much time, it shall return with a partial count or *errno* set to [EWOULDBLOCK] if no data were sent. The default for this option is the value zero, which indicates that a send operation will not timeout. It is implementation-defined whether the SO_SNDTIMEO option can be set.

The SO_TYPE option is used only on *getsockopt*( ). When this option is specified, *getsockopt*( ) shall return the type of the socket (for example, SOCK_STREAM). This option is useful to servers that inherit sockets on start-up. SO_TYPE has no default value.

2.10.17 Use of Sockets for Local UNIX Connections

Support for UNIX domain sockets is mandatory.

UNIX domain sockets provide process-to-process communication in a single system.

2.10.17.1 Headers

The symbolic constant AF_UNIX defined in the **<sys/socket.h>** header is used to identify the UNIX domain address family. The **<sys/un.h>** header contains other definitions used in connection with UNIX domain sockets. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to accommodate a **sockaddr_un** structure (see the **<sys/un.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_un** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_un** structure, the *ss_family* field maps onto the *sun_family* field.

2.10.18 Use of Sockets over Internet Protocols

When a socket is created in the Internet family with a protocol value of zero, the implementation shall use the protocol listed below for the type of socket created.

	SOCK_STREAM	IPPROTO_TCP.
	SOCK_DGRAM	IPPROTO_UDP.
RS	SOCK_RAW	IPPROTO_RAW.
	SOCK_SEQPACKET	Unspecified.

RS A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The default protocol for type SOCK_RAW shall be identified in the IP header with the value IPPROTO_RAW. Applications should not use the default protocol when creating a socket with type SOCK_RAW, but should identify a specific protocol by value. The ICMP control protocol is accessible from a raw socket by specifying a value of IPPROTO_ICMP for protocol.

2.10.19 Use of Sockets over Internet Protocols Based on IPv4

Support for sockets over Internet protocols based on IPv4 is mandatory.

2.10.19.1 Headers

The symbolic constant AF_INET defined in the **<sys/socket.h>** header is used to identify the IPv4 Internet address family. The **<netinet/in.h>** header contains other definitions used in connection with IPv4 Internet sockets. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to accommodate a **sockaddr_in** structure (see the **<netinet/in.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_in** structure, the *ss_family* field maps onto the *sin_family* field.

2.10.20 Use of Sockets over Internet Protocols Based on IPv6

IP6 This section describes extensions to support sockets over Internet protocols based on IPv6. This functionality is dependent on support of the IPV6 option (and the rest of this section is not further shaded for this option).

To enable smooth transition from IPv4 to IPv6, the features defined in this section may, in certain circumstances, also be used in connection with IPv4; see Section 2.10.20.2 (on page 518).

2.10.20.1 Addressing

IPv6 overcomes the addressing limitations of previous versions by using 128-bit addresses instead of 32-bit addresses. The IPv6 address architecture is described in RFC 2373.

There are three kinds of IPv6 address:

Unicast

Identifies a single interface.

A unicast address can be global, link-local (designed for use on a single link), or site-local (designed for systems not connected to the Internet). Link-local and site-local addresses need not be globally unique.

Anycast

Identifies a set of interfaces such that a packet sent to the address can be delivered to any member of the set.

An anycast address is similar to a unicast address; the nodes to which an anycast address is assigned must be explicitly configured to know that it is an anycast address.

Multicast

Identifies a set of interfaces such that a packet sent to the address should be delivered to every member of the set.

An application can send multicast datagrams by simply specifying an IPv6 multicast address in the *address* argument of *sendto*( ). To receive multicast datagrams, an application must join the multicast group (using *setsockopt*( ) with IPV6_JOIN_GROUP) and must bind to the socket the UDP port on which datagrams will be received. Some applications should also bind the multicast group address to the socket, to prevent other datagrams destined to that port from being delivered to the socket.

A multicast address can be global, node-local, link-local, site-local, or organization-local.

The following special IPv6 addresses are defined:

Unspecified
An address that is not assigned to any interface and is used to indicate the absence of an address.

Loopback
A unicast address that is not assigned to any interface and can be used by a node to send packets to itself.

Two sets of IPv6 addresses are defined to correspond to IPv4 addresses:

IPv4-compatible addresses
These are assigned to nodes that support IPv6 and can be used when traffic is ‘‘tunneled’’ through IPv4.

IPv4-mapped addresses
These are used to represent IPv4 addresses in IPv6 address format; see Section 2.10.20.2.

Note that the unspecified address and the loopback address must not be treated as IPv4-compatible addresses.

2.10.20.2 Compatibility with IPv4

The API provides the ability for IPv6 applications to interoperate with applications using IPv4, by using IPv4-mapped IPv6 addresses. These addresses can be generated automatically by the *getaddrinfo*( ) function when the specified host has only IPv4 addresses.

Applications may use AF_INET6 sockets to open TCP connections to IPv4 nodes, or send UDP packets to IPv4 nodes, by simply encoding the destination’s IPv4 address as an IPv4-mapped IPv6 address, and passing that address, within a **sockaddr_in6** structure, in the *connect*( ), *sendto*( ) or *sendmsg*( ) function. When applications use AF_INET6 sockets to accept TCP connections from IPv4 nodes, or receive UDP packets from IPv4 nodes, the system shall return the peer’s address to the application in the *accept*( ), *recvfrom*( ), *recvmsg*( ), or *getpeername*( ) function using a **sockaddr_in6** structure encoded this way. If a node has an IPv4 address, then the implementation may allow applications to communicate using that address via an AF_INET6 socket. In such a case, the address will be represented at the API by the corresponding IPv4-mapped IPv6 address. Also, the implementation may allow an AF_INET6 socket bound to **in6addr_any** to receive inbound connections and packets destined to one of the node’s IPv4 addresses.

An application may use AF_INET6 sockets to bind to a node’s IPv4 address by specifying the address as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure in the *bind*( ) function. For an AF_INET6 socket bound to a node’s IPv4 address, the system shall return the address in the *getsockname*( ) function as an IPv4-mapped IPv6 address in a **sockaddr_in6** structure.

2.10.20.3 Interface Identification

Each local interface is assigned a unique positive integer as a numeric index. Indexes start at 1; zero is not used. There may be gaps so that there is no current interface for a particular positive index. Each interface also has a unique implementation-defined name.

2.10.20.4 Options

The following options apply at the IPPROTO_IPV6 level:

IPV6_JOIN_GROUP

When set via *setsockopt*( ), it joins the application to a multicast group on an interface (identified by its index) and addressed by a given multicast address, enabling packets sent to that address to be read via the socket. If the interface index is specified as zero, the system selects the interface (for example, by looking up the address in a routing table and using the resulting interface).

An attempt to read this option using *getsockopt*( ) shall result in an [EOPNOTSUPP] error.

The value of this option is an **ipv6_mreq** structure.

IPV6_LEAVE_GROUP

When set via *setsockopt*( ), it removes the application from the multicast group on an interface (identified by its index) and addressed by a given multicast address.

An attempt to read this option using *getsockopt*( ) shall result in an [EOPNOTSUPP] error.

The value of this option is an **ipv6_mreq** structure.

IPV6_MULTICAST_HOPS

The value of this option is the hop limit for outgoing multicast IPv6 packets sent via the socket. Its possible values are the same as those of IPV6_UNICAST_HOPS. If the IPV6_MULTICAST_HOPS option is not set, a value of 1 is assumed. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

IPV6_MULTICAST_IF

The index of the interface to be used for outgoing multicast packets. It can be set via *setsockopt*( ) and read via *getsockopt*( ).

IPV6_MULTICAST_LOOP

This option controls whether outgoing multicast packets should be delivered back to the local application when the sending interface is itself a member of the destination multicast group. If it is set to 1 they are delivered. If it is set to 0 they are not. Other values result in an [EINVAL] error. This option can be set via *setsockopt*( ) and read via *getsockopt*( ).

IPV6_UNICAST_HOPS

The value of this option is the hop limit for outgoing unicast IPv6 packets sent via the socket. If the option is not set, or is set to −1, the system selects a default value. Attempts to set a value less than −1 or greater than 255 shall result in an [EINVAL] error. This option can be set via *setsockopt*( ) and read via *getsockopt*( ). |

IPV6_V6ONLY |

This socket option restricts AF_INET6 sockets to IPv6 communications only. AF_INET6 sockets may be used for both IPv4 and IPv6 communications. Some applications may want to restrict their use of an AF_INET6 socket to IPv6 communications only. For these applications, the IPv6_V6ONLY socket option is defined. When this option is turned on, the socket can be used to send and receive IPv6 packets only. This is an IPPROTO_IPV6-level option. This option takes an **int** value. This is a Boolean option. By default, this option is turned off. |

An [EOPNOTSUPP] error shall result if IPV6_JOIN_GROUP or IPV6_LEAVE_GROUP is used with *getsockopt*( ).

2.10.20.5 Headers

The symbolic constant AF_INET6 is defined in the **<sys/socket.h>** header to identify the IPv6 Internet address family. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

The **sockaddr_storage** structure defined in **<sys/socket.h>** shall be large enough to accommodate a **sockaddr_in6** structure (see the **<netinet/in.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) and shall be aligned at an appropriate boundary so that pointers to it can be cast as pointers to **sockaddr_in6** structures and used to access the fields of those structures without alignment problems. When a **sockaddr_storage** structure is cast as a **sockaddr_in6** structure, the *ss_family* field maps onto the *sin6_family* field.

The **<netinet/in.h>**, **<arpa/inet.h>**, and **<netdb.h>** headers contain other definitions used in connection with IPv6 Internet sockets; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers.

2.11 Tracing

TRC This section describes extensions to support tracing of user applications. This functionality is dependent on support of the Trace option (and the rest of this section is not further shaded for this option).

The tracing facilities defined in IEEE Std 1003.1-200x allow a process to select a set of trace event types, to activate a trace stream of the selected trace events as they occur in the flow of execution, and to retrieve the recorded trace events.

The tracing operation relies on three logically different components: the traced process, the controller process, and the analyzer process. During the execution of the traced process, when a trace point is reached, a trace event is recorded into the trace streams created for that process in which the associated trace event type identifier is not being filtered out. The controller process controls the operation of recording the trace events into the trace stream. It shall be able to:

- Initialize the attributes of a trace stream
- Create the trace stream (for a specified traced process) using those attributes
- Start and stop tracing for the trace stream
- Filter the type of trace events to be recorded, if the Trace Event Filter option is supported
- Shut a trace stream down

These operations can be done for an active trace stream. The analyzer process retrieves the traced events either at runtime, when the trace stream has not yet been shut down, but is still recording trace events; or after opening a trace log that had been previously recorded and shut down. These three logically different operations can be performed by the same process, or can be distributed into different processes.

A trace stream identifier can be created by a call to *posix_trace_create*( ), *posix_trace_create_withlog*( ), or *posix_trace_open*( ). The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions should be used by a controller process. The *posix_trace_open*( ) should be used by an analyzer process.

The tracing functions can serve different purposes. One purpose is debugging the possibly pre-instrumented code, while another is post-mortem fault analysis. These two potential uses differ in that the first requires pre-filtering capabilities to avoid overwhelming the trace stream and permits focusing on expected information; while the second needs comprehensive trace capabilities in order to be able to record all types of information.

The events to be traced belong to two classes:

1. User trace events (generated by the application instrumentation)
2. System trace events (generated by the operating system)

The trace interface defines several system trace event types associated with control of and operation of the trace stream. This small set of system trace events includes the minimum required to interpret correctly the trace event information present in the stream. Other desirable system trace events for some particular application profile may be implemented and are encouraged; for example, process and thread scheduling, signal occurrence, and so on.

Each traced process shall have a mapping of the trace event names to trace event type identifiers that have been defined for that process. Each active trace stream shall have a mapping that incorporates all the trace event type identifiers predefined by the trace system plus all the mappings of trace event names to trace event type identifiers of the processes that are being traced into that trace stream. These mappings are defined from the instrumented application by calling the *posix_trace_eventid_open*( ) function and from the controller process by calling the *posix_trace_trid_eventid_open*( ) function. For a pre-recorded trace stream, the list of trace event types is obtained from the pre-recorded trace log.

The *st_ctime* and *st_mtime* fields of a file associated with an active trace stream shall be marked for update every time any of the tracing operations modifies that file.

The *st_atime* field of a file associated with a trace stream shall be marked for update every time any of the tracing operations causes data to be read from that file.

Results are undefined if the application performs any operation on a file descriptor associated with an active or pre-recorded trace stream until *posix_trace_shutdown*( ) or *posix_trace_close*( ) is called for that trace stream.

The main purpose of this option is to define a complete set of functions and concepts that allow a conforming application to be traced from creation to termination, whatever its realtime constraints and properties.

2.11.1 Tracing Data Definitions

2.11.1.1 Structures

The **<trace.h>** header shall define the *posix_trace_status_info* and *posix_trace_event_info* structures described below. Implementations may add extensions to these structures.

posix_trace_status_info Structure

To facilitate control of a trace stream, information about the current state of an active trace stream can be obtained dynamically. This structure is returned by a call to the *posix_trace_get_status*( ) function.

The **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following members:

Member Type	Member Name	Description
int	*posix_stream_status*	The operating mode of the trace stream.
int	*posix_stream_full_status*	The full status of the trace stream.
int	*posix_stream_overrun_status*	Indicates whether trace events were lost in the trace stream.

If the Trace Log option is supported in addition to the Trace option, the **posix_trace_status_info** structure defined in **<trace.h>** shall contain at least the following additional members:

Member Type	Member Name	Description
int	*posix_stream_flush_status*	Indicates whether a flush is in progress.
int	*posix_stream_flush_error*	Indicates whether any error occurred during the last flush operation.
int	*posix_log_overrun_status*	Indicates whether trace events were lost in the trace log.
int	*posix_log_full_status*	The full status of the trace log.

The *posix_stream_status* member indicates the operating mode of the trace stream and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_RUNNING
Tracing is in progress; that is, the trace stream is accepting trace events.

POSIX_TRACE_SUSPENDED
The trace stream is not accepting trace events. The tracing operation has not yet started or has stopped, either following a *posix_trace_stop*( ) function call or because the trace resources are exhausted.

The *posix_stream_full_status* member indicates the full status of the trace stream, and it shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_FULL
The space in the trace stream for trace events is exhausted.

POSIX_TRACE_NOT_FULL
There is still space available in the trace stream.

The combination of the *posix_stream_status* and *posix_stream_full_status* members also indicates the actual status of the stream. The status shall be interpreted as follows:

POSIX_TRACE_RUNNING and POSIX_TRACE_NOT_FULL
This status combination indicates that tracing is in progress, and there is space available for recording more trace events.

POSIX_TRACE_RUNNING and POSIX_TRACE_FULL
This status combination indicates that tracing is in progress and that the trace stream is full of trace events. This status combination cannot occur unless the *stream-full-policy* is set to POSIX_TRACE_LOOP. The trace stream contains trace events recorded during a moving time window of prior trace events, and some older trace events may have been overwritten and thus lost.

POSIX_TRACE_SUSPENDED and POSIX_TRACE_NOT_FULL
This status combination indicates that tracing has not yet been started, has been stopped by the *posix_trace_stop*( ) function, or has been cleared by the *posix_trace_clear*( ) function.

POSIX_TRACE_SUSPENDED and POSIX_TRACE_FULL

This status combination indicates that tracing has been stopped by the implementation because the *stream-full-policy* attribute was POSIX_TRACE_UNTIL_FULL and trace resources were exhausted, or that the trace stream was stopped by the function *posix_trace_stop*( ) at a time when trace resources were exhausted.

The *posix_stream_overrun_status* member indicates whether trace events were lost in the trace stream, and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_OVERRUN

At least one trace event was lost and thus was not recorded in the trace stream.

POSIX_TRACE_NO_OVERRUN

No trace events were lost.

When the corresponding trace stream is created, the *posix_stream_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN.

Whenever an overrun occurs, *posix_stream_overrun_status* member shall be set to POSIX_TRACE_OVERRUN.

An overrun occurs when:

- The policy is POSIX_TRACE_LOOP and a recorded trace event is overwritten.
- The policy is POSIX_TRACE_UNTIL_FULL and the trace stream is full when a trace event is generated.
- If the Trace Log option is supported, the policy is POSIX_TRACE_FLUSH and at least one trace event is lost while flushing the trace stream to the trace log.

The *posix_stream_overrun_status* member is reset to zero after its value is read.

If the Trace Log option is supported in addition to the Trace option, the *posix_stream_flush_status*, *posix_stream_flush_error*, *posix_log_overrun_status*, and *posix_log_full_status* members are defined as follows; otherwise, they are undefined.

The *posix_stream_flush_status* member indicates whether a flush operation is being performed and shall have one of the following values defined by manifest constants in the header **<trace.h>**:

POSIX_TRACE_FLUSHING

The trace stream is currently being flushed to the trace log.

POSIX_TRACE_NOT_FLUSHING

No flush operation is in progress.

The *posix_stream_flush_status* member shall be set to POSIX_TRACE_FLUSHING if a flush operation is in progress either due to a call to the *posix_trace_flush*( ) function (explicit or caused by a trace stream shutdown operation) or because the trace stream has become full with the *stream-full-policy* attribute set to POSIX_TRACE_FLUSH. The *posix_stream_flush_status* member shall be set to POSIX_TRACE_NOT_FLUSHING if no flush operation is in progress.

The *posix_stream_flush_error* member shall be set to zero if no error occurred during flushing. If an error occurred during a previous flushing operation, the *posix_stream_flush_error* member shall be set to the value of the first error that occurred. If more than one error occurs while flushing, error values after the first shall be discarded. The *posix_stream_flush_error* member is reset to zero after its value is read.

The *posix_log_overrun_status* member indicates whether trace events were lost in the trace log, and shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_OVERRUN
At least one trace event was lost.

POSIX_TRACE_NO_OVERRUN
No trace events were lost.

When the corresponding trace stream is created, the *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN. Whenever an overrun occurs, this status shall be set to POSIX_TRACE_OVERRUN. The *posix_log_overrun_status* member is reset to zero after its value is read.

The *posix_log_full_status* member indicates the full status of the trace log, and it shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_FULL
The space in the trace log is exhausted.

POSIX_TRACE_NOT_FULL
There is still space available in the trace log.

The *posix_log_full_status* member is only meaningful if the *log-full-policy* attribute is either POSIX_TRACE_UNTIL_FULL or POSIX_TRACE_LOOP.

For an active trace stream without log, that is created by the *posix_trace_create*( ) function, the *posix_log_overrun_status* member shall be set to POSIX_TRACE_NO_OVERRUN and the *posix_log_full_status* member shall be set to POSIX_TRACE_NOT_FULL.

posix_trace_event_info Structure

The trace event structure **posix_trace_event_info** contains the information for one recorded trace event. This structure is returned by the set of functions *posix_trace_getnext_event*( ), *posix_trace_timedgetnext_event*( ), and *posix_trace_trygetnext_event*( ).

The **posix_trace_event_info** structure defined in **<trace.h>** shall contain at least the following members:

Member Type	Member Name	Description
trace_event_id_t	*posix_event_id*	Trace event type identification.
pid_t	*posix_pid*	Process ID of the process that generated the trace event.
void *	*posix_prog_address*	Address at which the trace point was invoked.
int	*posix_truncation_status*	Status about the truncation of the data associated with this trace event.
struct timespec	*posix_timestamp*	Time at which the trace event was generated.

In addition, if the Trace option and the Threads option are both supported, the **posix_trace_event_info** structure defined in **<trace.h>** shall contain the following additional member:

Member Type	Member Name	Description
pthread_t	*posix_thread_id*	Thread ID of the thread that generated the trace event.

The *posix_event_id* member represents the identification of the trace event type and its value is not directly defined by the user. This identification is returned by a call to one of the following functions: *posix_trace_trid_eventid_open*( ), *posix_trace_eventtypelist_getnext_id*( ), or *posix_trace_eventid_open*( ). The name of the trace event type can be obtained by calling *posix_trace_eventid_get_name*( ).

The *posix_pid* is the process identifier of the traced process which generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any process, the *posix_pid* member shall be set to zero.

For a user trace event, the *posix_prog_address* member is the process mapped address of the point at which the associated call to the *posix_trace_event*( ) function was made. For a system trace event, if the trace event is caused by a system service explicitly called by the application, the *posix_prog_address* member shall be the address of the process at the point where the call to that system service was made.

The *posix_truncation_status* member defines whether the data associated with a trace event has been truncated at the time the trace event was generated, or at the time the trace event was read from the trace stream, or (if the Trace Log option is supported) from the trace log (see the *event* argument from the *posix_trace_getnext_event*( ) function). The *posix_truncation_status* member shall have one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_NOT_TRUNCATED
All the traced data is available.

POSIX_TRACE_TRUNCATED_RECORD
Data was truncated at the time the trace event was generated.

POSIX_TRACE_TRUNCATED_READ
Data was truncated at the time the trace event was read from a trace stream or a trace log because the reader's buffer was too small. This truncation status overrides the POSIX_TRACE_TRUNCATED_RECORD status.

The *posix_timestamp* member shall be the time at which the trace event was generated. The clock used is implementation-defined, but the resolution of this clock can be retrieved by a call to the *posix_trace_attr_getclockres*( ) function.

If the Threads option is supported in addition to the Trace option:

- The *posix_thread_id* member is the identifier of the thread that generated the trace event. If the *posix_event_id* member is one of the implementation-defined system trace events and that trace event is not associated with any thread, the *posix_thread_id* member shall be set to zero.

Otherwise, this member is undefined.

2.11.1.2 Trace Stream Attributes

Trace streams have attributes that compose the **posix_trace_attr_t** trace stream attributes object. This object shall contain at least the following attributes:

- The *generation-version* attribute identifies the origin and version of the trace system.

- The *trace-name* attribute is a character string defined by the trace controller, and that identifies the trace stream.
- The *creation-time* attribute represents the time of the creation of the trace stream.
- The *clock-resolution* attribute defines the clock resolution of the clock used to generate timestamps.
- The *stream-min-size* attribute defines the minimum size in bytes of the trace stream strictly reserved for the trace events.
- The *stream-full-policy* attribute defines the policy followed when the trace stream is full; its value is POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_FLUSH.
- The *max-data-size* attribute defines the maximum record size in bytes of a trace event.

In addition, if the Trace option and the Trace Inherit option are both supported, the **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following attributes:

- The *inheritance* attribute specifies whether a newly created trace stream will inherit tracing in its parent's process trace stream. It is either POSIX_TRACE_INHERITED or POSIX_TRACE_CLOSE_FOR_CHILD.

In addition, if the Trace option and the Trace Log option are both supported, the **posix_trace_attr_t** trace stream creation attributes object shall contain at least the following attribute:

- If the file type corresponding to the trace log supports the POSIX_TRACE_LOOP or the POSIX_TRACE_UNTIL_FULL policies, the *log-max-size* attribute defines the maximum size in bytes of the trace log associated with an active trace stream. Other stream data—for example, trace attribute values—shall not be included in this size.
- The *log-full-policy* attribute defines the policy of a trace log associated with an active trace stream to be POSIX_TRACE_LOOP, POSIX_TRACE_UNTIL_FULL, or POSIX_TRACE_APPEND.

2.11.2 Trace Event Type Definitions

2.11.2.1 System Trace Event Type Definitions

The following system trace event types, defined in the **<trace.h>** header, track the invocation of the trace operations:

- POSIX_TRACE_START shall be associated with a trace start operation.
- POSIX_TRACE_STOP shall be associated with a trace stop operation.
- if the Trace Event Filter option is supported, POSIX_TRACE_FILTER shall be associated with a trace event type filter change operation.

The following system trace event types, defined in the **<trace.h>** header, report operational trace events:

- POSIX_TRACE_OVERFLOW shall mark the beginning of a trace overflow condition.
- POSIX_TRACE_RESUME shall mark the end of a trace overflow condition.
- If the Trace Log option is supported, POSIX_TRACE_FLUSH_START shall mark the beginning of a flush operation.

- If the Trace Log option is supported, POSIX_TRACE_FLUSH_STOP shall mark the end of a flush operation.
- If an implementation-defined trace error condition is reported, it shall be marked POSIX_TRACE_ERROR.

The interpretation of a trace stream or a trace log by a trace analyzer process relies on the information recorded for each trace event, and also on system trace events that indicate the invocation of trace control operations and trace system operational trace events.

The POSIX_TRACE_START and POSIX_TRACE_STOP trace events specify the time windows during which the trace stream is running.

> The POSIX_TRACE_STOP trace event with an associated data that is equal to zero indicates a call of the function *posix_trace_stop*( ).
>
> The POSIX_TRACE_STOP trace event with an associated data that is different from zero indicates an automatic stop of the trace stream (see *posix_trace_attr_getstreamfullpolicy*( ) defined in the System Interfaces volume of IEEE Std 1003.1-200x).

The POSIX_TRACE_FILTER trace event indicates that a trace event type filter value changed while the trace stream was running.

The POSIX_TRACE_ERROR serves to inform the analyzer process that an implementation-defined internal error of the trace system occurred.

The POSIX_TRACE_OVERFLOW trace event shall be reported with a timestamp equal to the timestamp of the first trace event overwritten. This is an indication that some generated trace events have been lost.

The POSIX_TRACE_RESUME trace event shall be reported with a timestamp equal to the timestamp of the first valid trace event reported after the overflow condition ends and shall be reported before this first valid trace event. This is an indication that the trace system is reliably recording trace events after an overflow condition.

Each of these trace event types shall be defined by a constant trace event name and a **trace_event_id_t** constant; trace event data is associated with some of these trace events.

If the Trace option is supported and the Trace Event Filter option and the Trace Log option are not supported, the following predefined system trace events in Table 2-3 shall be defined:

Table 2-3 Trace Option: System Trace Events

Event Name	Constant	Associated Data Data Type
posix_trace_error	POSIX_TRACE_ERROR	error int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

If the Trace option and the Trace Event Filter option are both supported, and if the Trace Log option is not supported, the following predefined system trace events in Table 2-4 (on page 528) shall be defined:

Table 2-4 Trace and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error
		int
posix_trace_start	POSIX_TRACE_START	event_filter
		trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter
		trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.

If the Trace option and the Trace Log option are both supported, and if the Trace Event Filter option is not supported, the following predefined system trace events in Table 2-5 shall be defined:

Table 2-5 Trace and Trace Log Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error - int
posix_trace_start	POSIX_TRACE_START	None.
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

If the Trace option, the Trace Event Filter option, and the Trace Log option are all supported, the following predefined system trace events in Table 2-6 (on page 529) shall be defined:

Table 2-6 Trace, Trace Log, and Trace Event Filter Options: System Trace Events

Event Name	Constant	Associated Data
		Data Type
posix_trace_error	POSIX_TRACE_ERROR	error
		int
posix_trace_start	POSIX_TRACE_START	event_filter
		trace_event_set_t
posix_trace_stop	POSIX_TRACE_STOP	auto
		int
posix_trace_filter	POSIX_TRACE_FILTER	old_event_filter new_event_filter
		trace_event_set_t
posix_trace_overflow	POSIX_TRACE_OVERFLOW	None.
posix_trace_resume	POSIX_TRACE_RESUME	None.
posix_trace_flush_start	POSIX_TRACE_FLUSH_START	None.
posix_trace_flush_stop	POSIX_TRACE_FLUSH_STOP	None.

2.11.2.2 *User Trace Event Type Definitions*

The user trace event POSIX_TRACE_UNNAMED_USEREVENT is defined in the **<trace.h>** header. If the limit of per-process user trace event names represented by {TRACE_USER_EVENT_MAX} has already been reached, this predefined user event shall be returned when the application tries to register more events than allowed. The data associated with this trace event is application-defined.

The following predefined user trace event in Table 2-7 shall be defined:

Table 2-7 Trace Option: User Trace Event

Event Name	Constant
posix_trace_unnamed_userevent	POSIX_TRACE_UNNAMED_USEREVENT

2.11.3 Trace Functions

The trace interface is built and structured to improve portability through use of trace data of opaque type. The object-oriented approach for the manipulation of trace attributes and trace event type identifiers requires definition of many constructor and selector functions which operate on these opaque types. Also, the trace interface must support several different tracing roles. To facilitate reading the trace interface, the trace functions are grouped into small functional sets supporting the three different roles:

- A trace controller process requires functions to set up and customize all the resources needed to run a trace stream, including:
  - — Attribute initialization and destruction (*posix_trace_attr_init*( ))
  - — Identification information manipulation (*posix_trace_attr_getgenversion*( ))
  - — Trace system behavior modification (*posix_trace_attr_getinherited*( ))
  - — Trace stream and trace log size set (*posix_trace_attr_getmaxusereventsize*( ))

  - Trace stream creation, flush, and shutdown (*posix_trace_create*( ))
  - Trace stream and trace log clear (*posix_trace_clear*( ))
  - Trace event type identifier manipulation (*posix_trace_trid_eventid_open*( ))
  - Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*( ))
  - Trace event type set manipulation (*posix_trace_eventset_empty*( ))
  - Trace event type filter set (*posix_trace_set_filter*( ))
  - Trace stream start and stop (*posix_trace_start*( ))
  - Trace stream information and status read (*posix_trace_get_attr*( ))
- A traced process requires functions to instrument trace points:
  - Trace event type identifiers definition and trace points insertion (*posix_trace_event*( ))
- A trace analyzer process requires functions to retrieve information from a trace stream and trace log:
  - Identification information read (*posix_trace_attr_getgenversion*( ))
  - Trace system behavior information read (*posix_trace_attr_getinherited*( ))
  - Trace stream and trace log size get (*posix_trace_attr_getmaxusereventsize*( ))
  - Trace event type identifier manipulation (*posix_trace_trid_eventid_open*( ))
  - Trace event type identifier list exploration (*posix_trace_eventtypelist_getnext_id*( ))
  - Trace log open, rewind, and close (*posix_trace_open*( ))
  - Trace stream information and status read (*posix_trace_get_attr*( ))
  - trace event read (*posix_trace_getnext_event*( ))

2.12 Data Types

All of the data types used by various functions are defined by the implementation. The following table describes some of these types. Other types referenced in the description of a function, not mentioned here, can be found in the appropriate header for that function.

Defined Type	Description
cc_t	Type used for terminal special characters.
clock_t	Arithmetic type used for processor times, as defined in the ISO C standard.
clockid_t	Used for clock ID type in some timer functions.
dev_t	Arithmetic type used for device numbers.
DIR	Type representing a directory stream.
div_t	Structure type returned by the *div*( ) function.
FILE	Structure containing information about a file.
glob_t	Structure type used in pathname pattern matching.
fpos_t	Type containing all information needed to specify uniquely every

Defined Type	Description
	position within a file.
gid_t	Arithmetic type used for group IDs.
iconv_t	Type used for conversion descriptors.
id_t	Arithmetic type used as a general identifier; can be used to contain at least the largest of a **pid_t**, **uid_t**, or **gid_t**.
ino_t	Arithmetic type used for file serial numbers.
key_t	Arithmetic type used for XSI interprocess communication.
ldiv_t	Structure type returned by the *ldiv*( ) function.
mode_t	Arithmetic type used for file attributes.
mqd_t	Used for message queue descriptors.
nfds_t	Integer type used for the number of file descriptors.
nlink_t	Arithmetic type used for link counts.
off_t	Signed arithmetic type used for file sizes.
pid_t	Signed arithmetic type used for process and process group IDs.
pthread_attr_t	Used to identify a thread attribute object.
pthread_cond_t	Used for condition variables.
pthread_condattr_t	Used to identify a condition attribute object.
pthread_key_t	Used for thread-specific data keys.
pthread_mutex_t	Used for mutexes.
pthread_mutexattr_t	Used to identify a mutex attribute object.
pthread_once_t	Used for dynamic package initialization.
pthread_rwlock_t	Used for read-write locks.
pthread_rwlockattr_t	Used for read-write lock attributes.
pthread_t	Used to identify a thread.
ptrdiff_t	Signed integer type of the result of subtracting two pointers.
regex_t	Structure type used in regular expression matching.
regmatch_t	Structure type used in regular expression matching.
rlim_t	Unsigned arithmetic type used for limit values, to which objects of type **int** and **off_t** can be cast without loss of value.
sem_t	Type used in performing semaphore operations.
sig_atomic_t	Integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.
sigset_t	Integer or structure type of an object used to represent sets of signals.
size_t	Unsigned integer type used for size of objects.
speed_t	Type used for terminal baud rates.
ssize_t	Arithmetic type used for a count of bytes or an error indication.
suseconds_t	Signed arithmetic type used for time in microseconds.
tcflag_t	Type used for terminal modes.
time_t	Arithmetic type used for time in seconds, as defined in the ISO C standard.
timer_t	Used for timer ID returned by the *timer_create*( ) function.
uid_t	Arithmetic type used for user IDs.
useconds_t	Integer type used for time in microseconds.
va_list	Type used for traversing variable argument lists.
wchar_t	Integer type whose range of values can represent distinct codes for all members of the largest extended character set specified by the

Defined Type	Description
	supported locales.
wctype_t	Scalar type which represents a character class descriptor.
wint_t	Integer type capable of storing any valid value of **wchar_t** or WEOF.
wordexp_t	Structure type used in word expansion.

Chapter 3

System Interfaces

This chapter describes the functions, macros, and external variables to support applications portability at the C-language source level.

NAME

FD_CLR — macros for synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/time.h>

FD_CLR(int fd, fd_set *fdset);
FD_ISSET(int fd, fd_set *fdset);
FD_SET(int fd, fd_set *fdset);
FD_ZERO(fd_set *fdset);
```

DESCRIPTION

Refer to *pselect*( ).

NAME

_Exit, _exit — terminate a process

SYNOPSIS

```
#include <unistd.h>

void _Exit(int status);
void _exit(int status);
```

DESCRIPTION

Refer to *exit*( ).

NAME

_longjmp, _setjmp — non-local goto

SYNOPSIS

XSI

```
#include <setjmp.h>

void _longjmp(jmp_buf env, int val);
int _setjmp(jmp_buf env);
```

DESCRIPTION

The *_longjmp*( ) and *_setjmp*( ) functions shall be equivalent to *longjmp*( ) and *setjmp*( ), respectively, with the additional restriction that *_longjmp*( ) and *_setjmp*( ) shall not manipulate the signal mask. |

If *_longjmp*( ) is called even though *env* was never initialized by a call to *_setjmp*( ), or when the last such call was in a function that has since returned, the results are undefined.

RETURN VALUE

Refer to *longjmp*( ) and *setjmp*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If *_longjmp*( ) is executed and the environment in which *_setjmp*( ) was executed no longer exists, errors can occur. The conditions under which the environment of the *_setjmp*( ) no longer exists include exiting the function that contains the *_setjmp*( ) call, and exiting an inner block with temporary storage. This condition might not be detectable, in which case the *_longjmp*( ) occurs and, if the environment no longer exists, the contents of the temporary storage of an inner block are unpredictable. This condition might also cause unexpected process termination. If the function has returned, the results are undefined.

Passing *longjmp*( ) a pointer to a buffer not created by *setjmp*( ), passing *_longjmp*( ) a pointer to a buffer not created by *_setjmp*( ), passing *siglongjmp*( ) a pointer to a buffer not created by *sigsetjmp*( ), or passing any of these three functions a buffer that has been modified by the user can cause all the problems listed above, and more.

The *_longjmp*( ) and *_setjmp*( ) functions are included to support programs written to historical system interfaces. New applications should use *siglongjmp*( ) and *sigsetjmp*( ) respectively.

RATIONALE

None.

FUTURE DIRECTIONS

The *_longjmp*( ) and *_setjmp*( ) functions may be marked LEGACY in a future version.

SEE ALSO

longjmp( ), *setjmp*( ), *siglongjmp*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

_setjmp — set jump point for a non-local goto

SYNOPSIS

XSI

```
#include <setjmp.h>

int _setjmp(jmp_buf env);
```

DESCRIPTION

Refer to *_longjmp*( ).

NAME

_tolower — transliterate uppercase characters to lowercase

SYNOPSIS

XSI
```
#include <ctype.h>

int _tolower(int c);
```

DESCRIPTION

The *_tolower*( ) macro shall be equivalent to *tolower*(*c*) except that the application shall ensure that the argument *c* is an uppercase letter.

RETURN VALUE

Upon successful completion, *_tolower*( ) shall return the lowercase letter corresponding to the argument passed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tolower( ), *isupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

_toupper — transliterate lowercase characters to uppercase

SYNOPSIS

XSI
```
#include <ctype.h>

int _toupper(int c);
```

DESCRIPTION

The *_toupper*( ) macro shall be equivalent to *toupper*( ) except that the application shall ensure that the argument *c* is a lowercase letter.

RETURN VALUE

Upon successful completion, *_toupper*( ) shall return the uppercase letter corresponding to the argument passed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

islower( ), *toupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

a64l, l64a — convert between a 32-bit integer and a radix-64 ASCII string

SYNOPSIS

XSI

```
#include <stdlib.h>

long a64l(const char *s);
char *l64a(long value);
```

DESCRIPTION

These functions maintain numbers stored in radix-64 ASCII characters. This is a notation by which 32-bit integers can be represented by up to six characters; each character represents a digit in radix-64 notation. If the type **long** contains more than 32 bits, only the low-order 32 bits shall be used for these operations.

The characters used to represent digits are `'.'` (dot) for 0, `'/'` for 1, `'0'` through `'9'` for [2,11], `'A'` through `'Z'` for [12,37], and `'a'` through `'z'` for [38,63].

The *a64l*( ) function shall take a pointer to a radix-64 representation, in which the first digit is the least significant, and return the corresponding **long** value. If the string pointed to by *s* contains more than six characters, *a64l*( ) shall use the first six. If the first six characters of the string contain a null terminator, *a64l*( ) shall use only characters preceding the null terminator. The *a64l*( ) function shall scan the character string from left to right with the least significant digit on the left, decoding each character as a 6-bit radix-64 number. If the type **long** contains more than 32 bits, the resulting value is sign-extended. The behavior of *a64l*( ) is unspecified if *s* is a null pointer or the string pointed to by *s* was not generated by a previous call to *l64a*( ).

The *l64a*( ) function shall take a **long** argument and return a pointer to the corresponding radix-64 representation. The behavior of *l64a*( ) is unspecified if *value* is negative.

The value returned by *l64a*( ) may be a pointer into a static buffer. Subsequent calls to *l64a*( ) may overwrite the buffer.

The *l64a*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *a64l*( ) shall return the **long** value resulting from conversion of the input string. If a string pointed to by *s* is an empty string, *a64l*( ) shall return 0L.

The *l64a*( ) function shall return a pointer to the radix-64 representation. If *value* is 0L, *l64a*( ) shall return a pointer to an empty string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If the type **long** contains more than 32 bits, the result of *a64l*(*l64a*(*x*)) is *x* in the low-order 32 bits.

RATIONALE

This is not the same encoding as used by either encoding variant of the *uuencode* utility.

FUTURE DIRECTIONS

None.

SEE ALSO

strtoul( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *uuencode*

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

NAME

abort — generate an abnormal process abort

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *abort*( ) function shall cause abnormal process termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

CX The abnormal termination processing shall include at least the effect of *fclose*( ) on all open streams and the default actions defined for SIGABRT.

XSI On XSI-conformant systems, in addition the abnormal termination processing shall include the effect of *fclose*( ) on message catalog descriptors.

The SIGABRT signal shall be sent to the calling process as if by means of *raise*( ) with the argument SIGABRT.

CX The status made available to *wait*( ) or *waitpid*( ) by *abort*( ) shall be that of a process terminated by the SIGABRT signal. The *abort*( ) function shall override blocking or ignoring the SIGABRT signal.

RETURN VALUE

The *abort*( ) function shall not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Catching the signal is intended to provide the application writer with a portable means to abort processing, free from possible interference from any implementation-defined functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *kill*( ), *raise*( ), *signal*( ), *wait*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

abs — return an integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *abs*( ) function shall compute the absolute value of its integer operand, *i*. If the result cannot be represented, the behavior is undefined.

RETURN VALUE

The *abs*( ) function shall return the absolute value of its integer operand.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In two's-complement representation, the absolute value of the negative integer with largest magnitude {INT_MIN} might not be representable.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fabs( ), *labs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

accept — accept a new connection on a socket

SYNOPSIS

```
#include <sys/socket.h>

int accept(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

DESCRIPTION

The *accept*( ) function shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket.

The *accept*( ) function takes the following arguments:

socket	Specifies a socket that was created with *socket*( ), has been bound to an address with *bind*( ), and has issued a successful call to *listen*( ).
address	Either a null pointer, or a pointer to a **sockaddr** structure where the address of the connecting socket shall be returned.
address_len	Points to a **socklen_t** structure which on input specifies the length of the supplied **sockaddr** structure, and on output specifies the length of the stored address.

If *address* is not a null pointer, the address of the peer for the accepted connection shall be stored in the **sockaddr** structure pointed to by *address*, and the length of this address shall be stored in the object pointed to by *address_len*.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

If the listen queue is empty of connection requests and O_NONBLOCK is not set on the file descriptor for the socket, *accept*( ) shall block until a connection is present. If the *listen*( ) queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, *accept*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

RETURN VALUE

Upon successful completion, *accept*( ) shall return the non-negative file descriptor of the accepted socket. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *accept*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	O_NONBLOCK is set for the socket file descriptor and no connections are present to be accepted.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNABORTED]	A connection has been aborted.

[EINTR]	The *accept*( ) function was interrupted by a signal that was caught before a valid connection arrived.
[EINVAL]	The *socket* is not accepting connections.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum number of file descriptors in the system are already open.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The socket type of the specified socket does not support accepting connections.

The *accept*( ) function may fail if:

[ENOBUFS]	No buffer space is available.
[ENOMEM]	There was insufficient memory available to complete the operation.
XSR [EPROTO]	A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

EXAMPLES

None.

APPLICATION USAGE

When a connection is available, *select*( ) indicates that the file descriptor for the socket is ready for reading.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind( ), *connect*( ), *listen*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *accept*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

access — determine accessibility of a file

SYNOPSIS

```
#include <unistd.h>

int access(const char *path, int amode);
```

DESCRIPTION

The *access*( ) function shall check the file named by the pathname pointed to by the *path* argument for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

The value of *amode* is either the bitwise-inclusive OR of the access permissions to be checked (R_OK, W_OK, X_OK) or the existence test (F_OK).

If any access permissions are checked, each shall be checked individually, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions. If the process has appropriate privileges, an implementation may indicate success for X_OK even if none of the execute file permission bits are set.

RETURN VALUE

If the requested access is permitted, *access*( ) succeeds and shall return 0; otherwise, −1 shall be returned and *errno* shall be set to indicate the error.

ERRORS

The *access*( ) function shall fail if:

[EACCES]	Permission bits of the file mode do not permit the requested access, or search permission is denied on a component of the path prefix.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	Write access is requested for a file on a read-only file system.

The *access*( ) function may fail if:

[EINVAL]	The value of the *amode* argument is invalid.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
[ETXTBSY]	Write access is requested for a pure procedure (shared text) file that is being executed.

EXAMPLES

Testing for the Existence of a File

The following example tests whether a file named **myfile** exists in the **/tmp** directory.

```
#include <unistd.h>
...
int result;
const char *filename = "/tmp/myfile";

result = access (filename, F_OK);
```

APPLICATION USAGE

Additional values of *amode* other than the set defined in the description may be valid; for example, if a system has extended access controls.

RATIONALE

In early proposals, some inadequacies in the *access*( ) function led to the creation of an *eaccess*( ) function because:

1. Historical implementations of *access*( ) do not test file access correctly when the process' real user ID is superuser. In particular, they always return zero when testing execute permissions without regard to whether the file is executable.

2. The superuser has complete access to all files on a system. As a consequence, programs started by the superuser and switched to the effective user ID with lesser privileges cannot use *access*( ) to test their file access permissions.

However, the historical model of *eaccess*( ) does not resolve problem (1), so this volume of IEEE Std 1003.1-200x now allows *access*( ) to behave in the desired way because several implementations have corrected the problem. It was also argued that problem (2) is more easily solved by using *open*( ), *chdir*( ), or one of the *exec* functions as appropriate and responding to the error, rather than creating a new function that would not be as reliable. Therefore, *eaccess*( ) is not included in this volume of IEEE Std 1003.1-200x.

The sentence concerning appropriate privileges and execute permission bits reflects the two possibilities implemented by historical implementations when checking superuser access for X_OK.

New implementations are discouraged from returning X_OK unless at least one execution permission bit is set.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

acos, acosf, acosl — arc cosine functions

SYNOPSIS

```
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the principal value of the arc cosine of their argument *x*. The value of *x* should be in the range [−1,1].

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the arc cosine of *x*, in the range [0,π] radians.

MX For finite values of *x* not in the range [−1,1], a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is +1, +0 shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is finite and is not in the range [−1,1], or is ±Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cos( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *acosf*( ) and *acosl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

acosf — arc cosine functions

SYNOPSIS

```
#include <math.h>

float acosf(float x);
```

DESCRIPTION

Refer to *acos*( ).

NAME

acosh, acoshf, acoshl, — inverse hyperbolic cosine functions

SYNOPSIS

```
#include <math.h>

double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the inverse hyperbolic cosine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic cosine of their argument.

MX For finite values of $x < 1$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is +1, +0 shall be returned.

If *x* is +Inf, +Inf shall be returned.

If *x* is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is finite and less than +1.0, or is –Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cosh( ), *feclearexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *acosh*( ) function is no longer marked as an extension.

The *acoshf*( ), and *acoshl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

acosl — arc cosine functions

SYNOPSIS

```
#include <math.h>

long double acosl(long double x);
```

DESCRIPTION

Refer to *acos*( ).

NAME

aio_cancel — cancel an asynchronous I/O request (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_cancel*( ) function shall attempt to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, then all outstanding cancelable asynchronous I/O requests against *fildes* shall be canceled.

Normal asynchronous notification shall occur for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, then the normal asynchronous completion process shall take place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status shall be set to [ECANCELED] and the return status shall be −1. For requested operations that are not successfully canceled, the *aiocbp* shall not be modified by *aio_cancel*( ).

If *aiocbp* is not NULL, then if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

Which operations are cancelable is implementation-defined.

RETURN VALUE

The *aio_cancel*( ) function shall return the value AIO_CANCELED to the calling process if the requested operation(s) were canceled. The value AIO_NOTCANCELED shall be returned if at least one of the requested operation(s) cannot be canceled because it is in progress. In this case, the state of the other operations, if any, referenced in the call to *aio_cancel*( ) is not indicated by the return value of *aio_cancel*( ). The application may determine the state of affairs for these operations by using *aio_error*( ). The value AIO_ALLDONE is returned if all of the operations have already completed. Otherwise, the function shall return −1 and set *errno* to indicate the error.

ERRORS

The *aio_cancel*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.

EXAMPLES

None.

APPLICATION USAGE

The *aio_cancel*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

NAME

aio_error — retrieve errors status for an asynchronous I/O operation (**REALTIME**)

SYNOPSIS

AIO
```
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_error*( ) function shall return the error status associated with the **aiocb** structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the SIO *errno* value that would be set by the corresponding *read*( ), *write*( ), *fdatasync*( ), or *fsync*( ) operation. If the operation has not yet completed, then the error status shall be equal to [EINPROGRESS].

RETURN VALUE

If the asynchronous I/O operation has completed successfully, then 0 shall be returned. If the asynchronous operation has completed unsuccessfully, then the error status, as described for SIO *read*( ), *write*( ), *fdatasync*( ), and *fsync*( ), shall be returned. If the asynchronous I/O operation has not yet completed, then [EINPROGRESS] shall be returned.

ERRORS

The *aio_error*( ) function may fail if:

[EINVAL]	The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES

None.

APPLICATION USAGE

The *aio_error*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_fsync*( ), *aio_read*( ), *aio_return*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

NAME

aio_fsync — asynchronous file synchronization (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_fsync(int op, struct aiocb *aiocbp);

```

DESCRIPTION

The *aio_fsync*( ) function shall asynchronously force all I/O operations associated with the file indicated by the file descriptor *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument and queued at the time of the call to *aio_fsync*( ) to the synchronized I/O completion state. The function call shall return when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If *op* is O_DSYNC, all currently queued I/O operations shall be completed as if by a call to *fdatasync*( ); that is, as defined for synchronized I/O data integrity completion. If *op* is O_SYNC, all currently queued I/O operations shall be completed as if by a call to *fsync*( ); that is, as defined for synchronized I/O file integrity completion. If the *aio_fsync*( ) function fails, or if the operation queued by *aio_fsync*( ) fails, then, as for *fsync*( ) and *fdatasync*( ), outstanding I/O operations are not guaranteed to have been completed.

If *aio_fsync*( ) succeeds, then it is only the I/O that was queued at the time of the call to *aio_fsync*( ) that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The *aiocbp* argument refers to an asynchronous I/O control block. The *aiocbp* value may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is [EINPROGRESS]. When all data has been successfully transferred, the error status shall be reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation shall be set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to occur as specified in Section 2.4.1 (on page 478) when all operations have achieved synchronized I/O completion. All other members of the structure referenced by *aiocbp* are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If the *aio_fsync*( ) function fails or the *aiocbp* indicates an error condition, data is not guaranteed to have been successfully transferred.

RETURN VALUE

The *aio_fsync*( ) function shall return the value 0 to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value −1 and set *errno* to indicate the error.

ERRORS

The *aio_fsync*( ) function shall fail if:

[EAGAIN]	The requested asynchronous operation was not queued due to temporary resource limitations.
[EBADF]	The *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* argument is not a valid file descriptor open for writing.

[EINVAL]	This implementation does not support synchronized I/O for this file.
[EINVAL]	A value of *op* other than O_DSYNC or O_SYNC was specified.

In the event that any of the queued I/O operations fail, *aio_fsync*( ) shall return the error condition defined for *read*( ) and *write*( ). The error is returned in the error status for the asynchronous *fsync*( ) operation, which can be retrieved using *aio_error*( ).

EXAMPLES

None.

APPLICATION USAGE

The *aio_fsync*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl( ), *fdatasync*( ), *fsync*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

NAME

aio_read — asynchronous read from a file (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_read*( ) function shall read *aiocbp*->*aio_nbytes* from the file associated with *aiocbp*->*aio_fildes* into the buffer pointed to by *aiocbp*->*aio_buf*. The function call shall return when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to the scheduling priority of the process minus *aiocbp*->*aio_reqprio*.

The *aiocbp* value may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call shall return without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp*->*aio_lio_opcode* field shall be ignored by *aio_read*( ).

The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp*->*aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

SIO If synchronized I/O is enabled on the file associated with *aiocbp*->*aio_fildes*, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *aiocbp*->*aio_fildes*.

RETURN VALUE

The *aio_read*( ) function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value −1 and set *errno* to indicate the error.

ERRORS

The *aio_read*( ) function shall fail if:

[EAGAIN] The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to *aio_read*( ), or asynchronously. If any of the conditions below are detected synchronously, the *aio_read*( ) function shall return −1 and set *errno* to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to −1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF]	The *aiocbp*->*aio_fildes* argument is not a valid file descriptor open for reading.
[EINVAL]	The file offset value implied by *aiocbp*->*aio_offset* would be invalid, *aiocbp*->*aio_reqprio* is not a valid value, or *aiocbp*->*aio_nbytes* is an invalid value.

In the case that the *aio_read*( ) successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values normally returned by the *read*( ) function call. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the *read*( ) function call, or one of the following values:

[EBADF]	The *aiocbp*->*aio_fildes* argument is not a valid file descriptor open for reading.
[ECANCELED]	The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.
[EINVAL]	The file offset value implied by *aiocbp*->*aio_offset* would be invalid.

The following condition may be detected synchronously or asynchronously:

[EOVERFLOW]	The file is a regular file, *aiobcp*->*aio_nbytes* is greater than 0, and the starting offset in *aiobcp*->*aio_offset* is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aiocbp*->*aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_read*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *lio_listio*( ), *aio_return*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.

NAME

aio_return — retrieve return status of an asynchronous I/O operation (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

ssize_t aio_return(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_return*( ) function shall return the return status associated with the **aiocb** structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding *read*( ), *write*( ), or *fsync*( ) function call. If the error status for the operation is equal to [EINPROGRESS], then the return status for the operation is undefined. The *aio_return*( ) function may be called exactly once to retrieve the return status of a given asynchronous operation; thereafter, if the same **aiocb** structure is used in a call to *aio_return*( ) or *aio_error*( ), an error may be returned. When the **aiocb** structure referred to by *aiocbp* is used to submit another asynchronous operation, then *aio_return*( ) may be successfully used to retrieve the return status of that operation.

RETURN VALUE

If the asynchronous I/O operation has completed, then the return status, as described for *read*( ), *write*( ), and *fsync*( ), shall be returned. If the asynchronous I/O operation has not yet completed, the results of *aio_return*( ) are undefined.

ERRORS

The *aio_return*( ) function may fail if:

[EINVAL]	The *aiocbp* argument does not refer to an asynchronous operation whose return status has not yet been retrieved.

EXAMPLES

None.

APPLICATION USAGE

The *aio_return*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_fsync*( ), *aio_read*( ), *aio_write*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The [EINVAL] error condition is updated as a ‘‘may fail’’. This is for consistency with the DESCRIPTION.

NAME

aio_suspend — wait for an asynchronous I/O request (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_suspend(const struct aiocb * const list[], int nent,
    const struct timespec *timeout);
```

DESCRIPTION

The *aio_suspend*( ) function shall suspend the calling thread until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, until a signal interrupts the function, or, if *timeout* is not NULL, until the time interval specified by *timeout* has passed. If any of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to [EINPROGRESS]) at the time of the call, the function shall return without suspending the calling thread. The *list* argument is an array of pointers to asynchronous I/O control blocks. The *nent* argument indicates the number of elements in the array. Each **aiocb** structure pointed to has been used in initiating an asynchronous I/O request via *aio_read*( ), *aio_write*( ), or *lio_listio*( ). This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* are completed, then *aio_suspend*( ) shall return with an error. MON If the Monotonic Clock option is supported, the clock that shall be used to measure this time interval shall be the CLOCK_MONOTONIC clock.

RETURN VALUE

If the *aio_suspend*( ) function returns after one or more asynchronous I/O operations have completed, the function shall return zero. Otherwise, the function shall return a value of −1 and set *errno* to indicate the error.

The application may determine which asynchronous I/O completed by scanning the associated error and return status using *aio_error*( ) and *aio_return*( ), respectively.

ERRORS

The *aio_suspend*( ) function shall fail if:

[EAGAIN]	No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.
[EINTR]	A signal interrupted the *aio_suspend*( ) function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

EXAMPLES

None.

APPLICATION USAGE

The *aio_suspend*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), *lio_listio*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used.

NAME

aio_write — asynchronous write to a file (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

DESCRIPTION

The *aio_write*( ) function shall write *aiocbp*->*aio_nbytes* to the file associated with *aiocbp*->*aio_fildes* from the buffer pointed to by *aiocbp*->*aio_buf*. The function shall return when the write request has been initiated or, at a minimum, queued to the file or device.

PIO If prioritized I/O is supported for this file, then the asynchronous operation shall be submitted at a priority equal to the scheduling priority of the process minus *aiocbp*->*aio_reqprio*.

The *aiocbp* may be used as an argument to *aio_error*( ) and *aio_return*( ) in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The *aiocbp* argument points to an **aiocb** structure. If the buffer pointed to by *aiocbp*->*aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, then the behavior is undefined.

If O_APPEND is not set for the file descriptor *aio_fildes*, then the requested operation shall take place at the absolute position in the file as given by *aio_offset*, as if *lseek*( ) were called immediately prior to the operation with an *offset* equal to *aio_offset* and a *whence* equal to SEEK_SET. If O_APPEND is set for the file descriptor, write operations append to the file in the same order as the calls were made. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aiocbp*->*aio_lio_opcode* field shall be ignored by *aio_write*( ).

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

SIO If synchronized I/O is enabled on the file associated with *aiocbp*->*aio_fildes*, the behavior of this function shall be according to the definitions of synchronized I/O data integrity completion, and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding to the address range being changed, the result of that action is undefined.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *aiocbp*->*aio_fildes*.

RETURN VALUE

The *aio_write*( ) function shall return the value zero to the calling process if the I/O operation is successfully queued; otherwise, the function shall return the value −1 and set *errno* to indicate the error.

ERRORS

The *aio_write*( ) function shall fail if:

[EAGAIN] The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to *aio_write*( ), or asynchronously. If any of the conditions below are detected synchronously, the *aio_write*( ) function shall return −1 and set *errno* to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation shall be set to −1, and the error status of the asynchronous operation is set to the corresponding value.

[EBADF]	The *aiocbp*->*aio_fildes* argument is not a valid file descriptor open for writing.
[EINVAL]	The file offset value implied by *aiocbp*->*aio_offset* would be invalid, *aiocbp*->*aio_reqprio* is not a valid value, or *aiocbp*->*aio_nbytes* is an invalid value.

In the case that the *aio_write*( ) successfully queues the I/O operation, the return status of the asynchronous operation shall be one of the values normally returned by the *write*( ) function call. If the operation is successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the *write*( ) function call, or one of the following:

[EBADF]	The *aiocbp*->*aio_fildes* argument is not a valid file descriptor open for writing.
[EINVAL]	The file offset value implied by *aiocbp*->*aio_offset* would be invalid.
[ECANCELED]	The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.

The following condition may be detected synchronously or asynchronously:

[EFBIG]	The file is a regular file, *aiobcp*->*aio_nbytes* is greater than 0, and the starting offset in *aiobcp*->*aio_offset* is at or beyond the offset maximum in the open file description associated with *aiocbp*->*aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

The *aio_write*( ) function is part of the Asynchronous Input and Output option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_cancel( ), *aio_error*( ), *aio_read*( ), *aio_return*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lio_listio*( ), *lseek*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with *aiocbp*->*aio_fildes*.

- The [EFBIG] error is added as part of the large file support extensions.

NAME

alarm — schedule an alarm signal

SYNOPSIS

```
#include <unistd.h>

unsigned alarm(unsigned seconds);
```

DESCRIPTION

The *alarm*( ) function shall cause the system to generate a SIGALRM signal for the process after the number of realtime seconds specified by *seconds* have elapsed. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.

If *seconds* is 0, a pending alarm request, if any, is canceled.

Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call shall result in rescheduling the time at which the SIGALRM signal is generated.

XSI Interactions between *alarm*( ) and any of *setitimer*( ), *ualarm*( ), or *usleep*( ) are unspecified.

RETURN VALUE

If there is a previous *alarm*( ) request with time remaining, *alarm*( ) shall return a non-zero value that is the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, *alarm*( ) shall return 0.

ERRORS

The *alarm*( ) function is always successful, and no return value is reserved to indicate an error.

EXAMPLES

None.

APPLICATION USAGE

The *fork*( ) function clears pending alarms in the child process. A new process image created by one of the *exec* functions inherits the time left to an alarm signal in the old process' image.

Application writers should note that the type of the argument *seconds* and the return value of *alarm*( ) is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the ISO C standard sets as 65 535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit **int** type, consistently use either **unsigned** or **int**.

Application writers should be aware of possible interactions when the same process uses both the *alarm*( ) and *sleep*( ) functions.

RATIONALE

Many historical implementations (including Version 7 and System V) allow an alarm to occur up to a second early. Other implementations allow alarms up to half a second or one clock tick early or do not allow them to occur early at all. The latter is considered most appropriate, since it gives the most predictable behavior, especially since the signal can always be delayed for an indefinite amount of time due to scheduling. Applications can thus choose the *seconds* argument as the minimum amount of time they wish to have elapse before the signal.

The term *realtime* here and elsewhere (*sleep*( ), *times*( )) is intended to mean ''wall clock'' time as common English usage, and has nothing to do with ''realtime operating systems''. It is in contrast to *virtual time*, which could be misinterpreted if just *time* were used.

In some implementations, including 4.3 BSD, very large values of the *seconds* argument are silently rounded down to an implementation-defined maximum value. This maximum is large enough (on the order of several months) that the effect is not noticeable.

There were two possible choices for alarm generation in multi-threaded applications: generation for the calling thread or generation for the process. The first option would not have been particularly useful since the alarm state is maintained on a per-process basis and the alarm that is established by the last invocation of *alarm*( ) is the only one that would be active.

Furthermore, allowing generation of an asynchronous signal for a thread would have introduced an exception to the overall signal model. This requires a compelling reason in order to be justified.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *exec*, *fork*( ), *getitimer*( ), *pause*( ), *sigaction*( ), *sleep*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that interactions with the *setitimer*( ), *ualarm*( ), and *usleep*( ) functions are unspecified.

NAME

asctime, asctime_r — convert date and time to a string

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr);
char *asctime_r(const struct tm *restrict tm, char *restrict buf);
```

TSF (asctime_r)

DESCRIPTION

CX For *asctime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *asctime*( ) function shall convert the broken-down time in the structure pointed to by *timeptr* into a string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm:

```
char *asctime(const struct tm *timeptr)
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
        wday_name[timeptr->tm_wday],
        mon_name[timeptr->tm_mon],
        timeptr->tm_mday, timeptr->tm_hour,
        timeptr->tm_min, timeptr->tm_sec,
        1900 + timeptr->tm_year);
    return result;
}
```

The **tm** structure is defined in the **<time.h>** header.

CX The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The *asctime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *asctime_r*( ) function shall convert the broken-down time in the structure pointed to by *tm* into a string (of the same form as that returned by *asctime*( )) that is placed in the user-supplied buffer pointed to by *buf* (which shall contain at least 26 bytes) and then return *buf*.

RETURN VALUE

Upon successful completion, *asctime*( ) shall return a pointer to the string.

TSF Upon successful completion, *asctime_r*( ) shall return a pointer to a character string containing the date and time. This string is pointed to by the argument *buf*. If the function is unsuccessful, it shall return NULL.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ). This function is included for compatibility with older implementations, and does not support localized date and time formats. Applications should use *strftime*( ) to achieve maximum portability.

The *asctime_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

The *asctime_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *asctime*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *asctime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The DESCRIPTION of *asctime_r*( ) is updated to describe the format of the string returned.

The **restrict** keyword is added to the *asctime_r*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

asin, asinf, asinl — arc sine function

SYNOPSIS

```
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the principal value of the arc sine of their argument *x*. The value of *x* should be in the range [−1,1].

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the arc sine of *x*, in the range $[-\pi/2,\pi/2]$ radians.

MX For finite values of *x* not in the range [−1,1], a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

MX Domain Error	The *x* argument is finite and is not in the range [−1,1], or is ±Inf. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

These functions may fail if:

MX Range Error	The value of *x* is subnormal. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *sin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *asinf*( ) and *asinl*( ) functions are added for alignment with the ISO/IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

NAME

asinf — arc sine function

SYNOPSIS

```
#include <math.h>

float asinf(float x);
```

DESCRIPTION

Refer to *asin*( ).

NAME

asinfh, asinfl — inverse hyperbolic sine functions

SYNOPSIS

```
#include <math.h>

float asinhf(float x);
long double asinhl(long double x);
```

DESCRIPTION

Refer to *asinh*( ).

NAME

asinh, asinfh, asinfl — inverse hyperbolic sine functions

SYNOPSIS

```
#include <math.h>

double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the inverse hyperbolic sine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic sine, of their argument.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions may fail if:

MX Range Error	The value of *x* is subnormal. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *sinh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *asinh*( ) function is no longer marked as an extension.

The *asinhf*( ), and *asinhl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

asinl — arc sine function

SYNOPSIS

```
#include <math.h>
```

```
long double asinl(long double x);
```

DESCRIPTION

Refer to *asin*( ).

NAME

assert — insert program diagnostics

SYNOPSIS

```
#include <assert.h>

void assert(scalar expression);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *assert*( ) macro shall insert diagnostics into programs; it shall expand to a **void** expression. When it is executed, if *expression* (which shall have a **scalar** type) is false (that is, compares equal to 0), *assert*( ) shall write information about the particular call that failed on *stderr* and shall call *abort*( ). |

The information written about the call that failed shall include the text of the argument, the name of the source file, the source file line number, and the name of the enclosing function, the latter are, respectively, the values of the preprocessing macros __FILE__ and __LINE__ and of the identifier __func__.

Forcing a definition of the name NDEBUG, either from the compiler command line or with the preprocessor control statement **#define** NDEBUG ahead of the **#include <assert.h>** statement, shall stop assertions from being compiled into the program.

RETURN VALUE

The *assert*( ) macro shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abort( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<assert.h>**, *stderr*

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The prototype for the *expression* argument to *assert*( ) is changed from **int** to **scalar** for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION of *assert*( ) is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

atan, atanf, atanl — arc tangent function

SYNOPSIS

```
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the principal value of the arc tangent of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the arc tangent of *x* in the range [−π/2,π/2] radians.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0 *x* shall be returned.

If *x* is ±Inf, ±π/2 shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions may fail if:

MX Range Error The value of *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atan2( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *atanf*( ) and *atanl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

atan2, atan2f, atan2l — arc tangent functions

SYNOPSIS

```
#include <math.h>

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the arc tangent of y/x in the range $[-\pi,\pi]$ radians.

If y is ±0 and x is < 0, $\pm\pi$ shall be returned.

If y is ±0 and x is > 0, ±0 shall be returned.

If y is < 0 and x is ±0, $-\pi/2$ shall be returned.

If y is > 0 and x is ±0, $\pi/2$ shall be returned.

If x is 0, a pole error shall not occur.

MX If either x or y is NaN, a NaN shall be returned.

If the result underflows, a range error may occur and y/x should be returned.

If y is ±0 and x is −0, $\pm\pi$ shall be returned.

If y is ±0 and x is +0, ±0 shall be returned.

For finite values of $\pm y > 0$, if x is −Inf, $\pm\pi$ shall be returned.

For finite values of $\pm y > 0$, if x is +Inf, ±0 shall be returned.

For finite values of x, if y is ±Inf, $\pm\pi/2$ shall be returned.

If y is ±Inf and x is −Inf, $\pm 3\pi/4$ shall be returned.

If y is ±Inf and x is +Inf, $\pm\pi/4$ shall be returned.

If both arguments are 0, a domain error shall not occur.

ERRORS

These functions may fail if:

MX

Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression

(math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atan( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *atan2f*( ) and *atan2l*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

atanf — arc tangent function

SYNOPSIS

```
#include <math.h>

float atanf(float x);
```

DESCRIPTION

Refer to *atan*( ).

NAME

atanh, atanhf, atanhl — inverse hyperbolic tangent functions

SYNOPSIS

```
#include <math.h>

double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the inverse hyperbolic tangent of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the inverse hyperbolic tangent of their argument.

If *x* is ±1, a pole error shall occur, and *atanh*( ), *atanhf*( ), and *atanhl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively, with the same sign as the correct value of the function.

MX For finite |*x*|>1, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is finite and not in the range [−1,1], or is ±Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised. |

Pole Error The *x* argument is ±1.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised. |

These functions may fail if:

MX **Range Error** The value of *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *tanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *atanh*( ) function is no longer marked as an extension.

The *atanhf*( ), and *atanhl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

atanl — arc tangent function

SYNOPSIS

```
#include <math.h>

long double atanl(long double x);
```

DESCRIPTION

Refer to *atan*( ).

NAME

atexit — register a function to run at process termination

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *atexit*( ) function shall register the function pointed to by *func*, to be called without arguments at normal program termination. At normal program termination, all functions registered by the *atexit*( ) function shall be called, in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Normal termination occurs either by a call to *exit*( ) or a return from *main*( ).

At least 32 functions can be registered with *atexit*( ).

CX After a successful call to any of the *exec* functions, any functions previously registered by *atexit*( ) shall no longer be registered.

RETURN VALUE

Upon successful completion, *atexit*( ) shall return 0; otherwise, it shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The functions registered by a call to *atexit*( ) must return to ensure that all registered functions are called.

The application should call *sysconf*( ) to obtain the value of {ATEXIT_MAX}, the number of functions that can be registered. There is no way for an application to tell how many functions have already been registered with *atexit*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

atof — convert a string to double-precision number

SYNOPSIS

```
#include <stdlib.h>

double atof(const char *str);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The call *atof*(*str*) shall be equivalent to:

```
strtod(str,(char **)NULL),
```

except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE

The *atof*( ) function shall return the converted value if the value can be represented.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *atof*( ) function is subsumed by *strtod*( ) but is retained because it is used extensively in existing code. If the number is not known to be in range, *strtod*( ) should be used because *atof*( ) is not required to perform any error checking.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtod( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

atoi — convert a string to an integer

SYNOPSIS

```
#include <stdlib.h>

int atoi(const char *str);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The call *atoi*(*str*) shall be equivalent to:

```
(int) strtol(str, (char **)NULL, 10)
```

except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE

The *atoi*( ) function shall return the converted value if the value can be represented.

ERRORS

No errors are defined.

EXAMPLES

Converting an Argument

The following example checks for proper usage of the program. If there is an argument and the decimal conversion of this argument (obtained using *atoi*( )) is greater than 0, then the program has a valid number of minutes to wait for an event.

```
#include <stdlib.h>
#include <stdio.h>
...
int minutes_to_event;
...
if (argc < 2 || ((minutes_to_event = atoi (argv[1]))) <= 0) {
   fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
}
...
```

APPLICATION USAGE

The *atoi*( ) function is subsumed by *strtol*( ) but is retained because it is used extensively in existing code. If the number is not known to be in range, *strtol*( ) should be used because *atoi*( ) is not required to perform any error checking.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtol( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

atol, atoll — convert a string to a long integer

SYNOPSIS

```
#include <stdlib.h>

long atol(const char *str);
long long atoll(const char *nptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The call *atol*(*str*) shall be equivalent to:

```
strtol(str, (char **)NULL, 10)
```

The call *atoll*(*str*) shall be equivalent to:

```
strtoll(nptr, (char **)NULL, 10)
```

except that the handling of errors may differ. If the value cannot be represented, the behavior is undefined.

RETURN VALUE

These functions shall return the converted value if the value can be represented.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *atol*( ) function is subsumed by *strtol*( ) but is retained because it is used extensively in existing code. If the number is not known to be in range, *strtol*( ) should be used because *atol*( ) is not required to perform any error checking.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtol( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *atoll*( ) function is added for alignment with the ISO/IEC 9899: 1999 standard.

NAME

basename — return the last component of a pathname

SYNOPSIS

XSI
```
#include <libgen.h>

char *basename(char *path);
```

DESCRIPTION

The *basename*( ) function shall take the pathname pointed to by *path* and return a pointer to the final component of the pathname, deleting any trailing `'/'` characters.

If the string consists entirely of the `'/'` character, *basename*( ) shall return a pointer to the string `"/"`. If the string is exactly `"//"`, it is implementation-defined whether `'/'` or `"//"` is returned.

If *path* is a null pointer or points to an empty string, *basename*( ) shall return a pointer to the string `"."`.

The *basename*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by a subsequent call to *basename*( ).

The *basename*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *basename*( ) function shall return a pointer to the final component of *path*.

ERRORS

No errors are defined.

EXAMPLES

Using basename( )

The following program fragment returns a pointer to the value *lib*, which is the base name of **/usr/lib**.

```
#include <libgen.h>
...
char *name = "/usr/lib";
char *base;

base = basename(name);
...
```

Sample Input and Output Strings for basename( )

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the *basename*( ) function.

Input String	Output String
`"/usr/lib"`	`"lib"`
`"/usr/"`	`"usr"`
`"/"`	`"/"`

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dirname( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *basename*

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

bcmp — memory operations (**LEGACY**)

SYNOPSIS

XSI

```
#include <strings.h>

int bcmp(const void *s1, const void *s2, size_t n);
```

DESCRIPTION

The *bcmp*( ) function shall compare the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*.

RETURN VALUE

The *bcmp*( ) function shall return 0 if *s1* and *s2* are identical; otherwise, it shall return non-zero. Both areas are assumed to be *n* bytes long. If the value of *n* is 0, *bcmp*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memcmp( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *bcmp*( ) as follows:

```
#define bcmp(b1,b2,len) memcmp((b1), (b2), (size_t)(len))
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memcmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

bcopy — memory operations (**LEGACY**)

SYNOPSIS

XSI

```
#include <strings.h>

void bcopy(const void *s1, void *s2, size_t n);
```

DESCRIPTION

The *bcopy*( ) function shall copy *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*.

The bytes are copied correctly even if the area pointed to by *s1* overlaps the area pointed to by *s2*.

RETURN VALUE

The *bcopy*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memmove( ) is preferred over this function.

The following are approximately equivalent (note the order of the arguments):

```
bcopy(s1,s2,n) ~= memmove(s2,s1,n)
```

For maximum portability, it is recommended to replace the function call to *bcopy*( ) as follows:

```
#define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void) 0)
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memmove( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

bind — bind a name to a socket

SYNOPSIS

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

DESCRIPTION

The *bind*( ) function shall assign a local socket address *address* to a socket identified by descriptor *socket* that has no local socket address assigned. Sockets created with the *socket*( ) function are initially unnamed; they are identified only by their address family.

The *bind*( ) function takes the following arguments:

socket	Specifies the file descriptor of the socket to be bound.
address	Points to a **sockaddr** structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.
address_len	Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

The socket specified by *socket* may require the process to have appropriate privileges to use the *bind*( ) function.

RETURN VALUE

Upon successful completion, *bind*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *bind*( ) function shall fail if:

[EADDRINUSE]	The specified address is already in use.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
[EBADF]	The *socket* argument is not a valid file descriptor.
[EINVAL]	The socket is already bound to an address, and the protocol does not support binding to a new address; or the socket has been shut down.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The socket type of the specified socket does not support binding to an address.

If the address family of the socket is AF_UNIX, then *bind*( ) shall fail if:

[EACCES]	A component of the path prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.

[EDESTADDRREQ] or [EISDIR]
The *address* argument is a null pointer.

[EIO] An I/O error occurred.

[ELOOP] A loop exists in symbolic links encountered during resolution of the pathname in *address*.

[ENAMETOOLONG]
A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

[ENOENT] A component of the pathname does not name an existing file or the pathname is an empty string.

[ENOTDIR] A component of the path prefix of the pathname in *address* is not a directory.

[EROFS] The name would reside on a read-only file system.

The *bind*( ) function may fail if:

[EACCES] The specified address is protected and the current user does not have permission to bind to it.

[EINVAL] The *address_len* argument is not a valid length for the address family.

[EISCONN] The socket is already connected.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in *address*.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

[ENOBUFS] Insufficient resources were available to complete the call.

EXAMPLES

None.

APPLICATION USAGE

An application program can retrieve the assigned socket name with the *getsockname*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *getsockname*( ), *listen*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

bsd_signal — simplified signal facilities

SYNOPSIS

OB XSI
```
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

The *bsd_signal*( ) function provides a partially compatible interface for programs written to historical system interfaces (see APPLICATION USAGE).

The function call *bsd_signal*(*sig*, *func*) shall be equivalent to the following:

```
void (*bsd_signal(int sig, void (*func)(int)))(int)
{
    struct sigaction act, oact;

    act.sa_handler = func;
    act.sa_flags = SA_RESTART;
    sigemptyset(&act.sa_mask);
    sigaddset(&act.sa_mask, sig);
    if (sigaction(sig, &act, &oact) == -1)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

The handler function should be declared:

```
void handler(int sig);
```

where *sig* is the signal number. The behavior is undefined if *func* is a function that takes more than one argument, or an argument of a different type.

RETURN VALUE

Upon successful completion, *bsd_signal*( ) shall return the previous action for *sig*. Otherwise, SIG_ERR shall be returned and *errno* shall be set to indicate the error.

ERRORS

Refer to *sigaction*( ).

EXAMPLES

None.

APPLICATION USAGE

This function is a direct replacement for the BSD *signal*( ) function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use *sigaction*( ). The *bsd_signal*( ) function differs from *signal*( ) in that the SA_RESTART flag is set and the SA_RESETHAND is clear when *bsd_signal*( ) is used. The state of these flags is not specified for *signal*( ).

It is recommended that new applications use the *sigaction*( ) function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sigaction( ), *sigaddset*( ), *sigemptyset*( ), *signal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked obsolescent.

NAME

bsearch — binary search a sorted table

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel,
    size_t width, int (*compar)(const void *, const void *));
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *bsearch*( ) function shall search an array of *nel* objects, the initial element of which is pointed to by *base*, for an element that matches the object pointed to by *key*. The size of each element in the array is specified by *width*.

The comparison function pointed to by *compar* shall be called with two arguments that point to the *key* object and to an array element, in that order.

The application shall ensure that the function returns an integer less than, equal to, or greater than 0 if the *key* object is considered, respectively, to be less than, to match, or to be greater than the array element. The application shall ensure that the array consists of all the elements that compare less than, all the elements that compare equal to, and all the elements that compare greater than the *key* object, in that order.

RETURN VALUE

The *bsearch*( ) function shall return a pointer to a matching member of the array, or a null pointer if no match is found. If two or more members compare equal, which member is returned is unspecified.

ERRORS

No errors are defined.

EXAMPLES

The example below searches a table containing pointers to nodes consisting of a string and its length. The table is ordered alphabetically on the string in the node pointed to by each entry.

The code fragment below reads in strings and either finds the corresponding node and prints out the string and its length, or prints an error message.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TABSIZE    1000

struct node {                  /* These are stored in the table. */
    char *string;
    int length;
};
struct node table[TABSIZE];    /* Table to be searched. */
    .
    .
    .
{
    struct node *node_ptr, node;
    /* routine to compare 2 nodes */
```

```
        int node_compare(const void *, const void *);
        char str_space[20];   /* Space to read string into. */
        .
        .
        .
        node.string = str_space;
        while (scanf("%s", node.string) != EOF) {
            node_ptr = (struct node *)bsearch((void *)(&node),
                   (void *)table, TABSIZE,
                   sizeof(struct node), node_compare);
            if (node_ptr != NULL) {
                (void)printf("string = %20s, length = %d\n",
                    node_ptr->string, node_ptr->length);
            } else {
                (void)printf("not found: %s\n", node.string);
            }
        }
    }
    /*
        This routine compares two nodes based on an
        alphabetical ordering of the string field.
    */
    int
    node_compare(const void *node1, const void *node2)
    {
        return strcoll(((const struct node *)node1)->string,
            ((const struct node *)node2)->string);
    }
```

APPLICATION USAGE

The pointers to the key and the element at the base of the table should be of type pointer-to-element.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

In practice, the array is usually sorted according to the comparison function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *lsearch*( ), *qsort*( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

btowc — single byte to wide character conversion

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t btowc(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *btowc*( ) function shall determine whether *c* constitutes a valid (one-byte) character in the initial shift state.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *btowc*( ) function shall return WEOF if *c* has the value EOF or if **(unsigned char)** *c* does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it shall return the wide-character representation of that character.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wctob( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

NAME

bzero — memory operations (**LEGACY**)

SYNOPSIS

XSI

```
#include <strings.h>

void bzero(void *s, size_t n);
```

DESCRIPTION

The *bzero*( ) function shall place *n* zero-valued bytes in the area pointed to by *s*.

RETURN VALUE

The *bzero*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

memset( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *bzero*( ) as follows:

```
#define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

memset( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

cabs, cabsf, cabsl — return a complex absolute value

SYNOPSIS

```
#include <complex.h>

double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex absolute value (also called norm, modulus, or magnitude) of *z*.

RETURN VALUE

These functions shall return the complex absolute value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

cacos, cacosf, cacosl — complex arc cosine functions

SYNOPSIS

```
#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc cosine of *z*, with branch cuts outside the interval [−1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccos( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cacosf — complex arc cosine functions

SYNOPSIS

```
#include <complex.h>

float complex cacosf(float complex z);
```

DESCRIPTION

Refer to *cacos*( ).

NAME

cacosh, cacoshf, cacoshl — complex arc hyperbolic cosine functions

SYNOPSIS

```
#include <complex.h>

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ccosh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

cacosl — complex arc cosine functions

SYNOPSIS

```
#include <complex.h>

long double complex cacosl(long double complex z);
```

DESCRIPTION

Refer to *cacos*( ).

NAME

calloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *calloc*( ) function shall allocate unused space for an array of *nelem* elements each of whose size in bytes is *elsize*. The space shall be initialized to all bits 0.

The order and contiguity of storage allocated by successive calls to *calloc*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: the value returned shall be either a null pointer or a unique pointer.

RETURN VALUE

Upon successful completion with both *nelem* and *elsize* non-zero, *calloc*( ) shall return a pointer to the allocated space. If either *nelem* or *elsize* is 0, then either a null pointer or a unique pointer value that can be successfully passed to *free*( ) shall be returned. Otherwise, it shall return a null CX pointer and set *errno* to indicate the error.

ERRORS

The *calloc*( ) function shall fail if:

CX [ENOMEM] Insufficient memory is available.

EXAMPLES

None.

APPLICATION USAGE

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), *realloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The setting of *errno* and the [ENOMEM] error condition are mandatory if an insufficient memory condition occurs.

NAME

carg, cargf, cargl — complex argument functions

SYNOPSIS

```
#include <complex.h>

double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the argument (also called phase angle) of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the value of the argument in the interval [−π, +π].

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cimag( ), *conj*( ), *cproj*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

casin, casinf, casinl — complex arc sine functions

SYNOPSIS

```
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc sine of *z*, with branch cuts outside the interval [−1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csin( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

casinf — complex arc sine functions

SYNOPSIS

```
#include <complex.h>

float complex casinf(float complex z);
```

DESCRIPTION

Refer to *casin*( ).

NAME

casinh, casinhf, casinhl — complex arc hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval [−*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

csinh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

casinl — complex arc sine functions

SYNOPSIS

```
#include <complex.h>

long double complex casinl(long double complex z);
```

DESCRIPTION

Refer to *casin*( ).

NAME

catan, catanf, catanl — complex arc tangent functions

SYNOPSIS

```
#include <complex.h>

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc tangent of *z*, with branch cuts outside the interval [−*i*, +*i*] along the imaginary axis.

RETURN VALUE

These functions shall return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctan( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

catanf — complex arc tangent functions

SYNOPSIS

```
#include <complex.h>

float complex catanf(float complex z);
```

DESCRIPTION

Refer to *catan*( ).

NAME

catanh, catanhf, catanhl — complex arc hyperbolic tangent functions

SYNOPSIS

```
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval [−1, +1] along the real axis.

RETURN VALUE

These functions shall return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctanh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

catanl — complex arc tangent functions

SYNOPSIS

```
#include <complex.h>

long double complex catanl(long double complex z);
```

DESCRIPTION

Refer to *catan*( ).

NAME

catclose — close a message catalog descriptor

SYNOPSIS

XSI

```
#include <nl_types.h>

int catclose(nl_catd catd);
```

DESCRIPTION

The *catclose*( ) function shall close the message catalog identified by *catd*. If a file descriptor is used to implement the type **nl_catd**, that file descriptor shall be closed.

RETURN VALUE

Upon successful completion, *catclose*( ) shall return 0; otherwise, −1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *catclose*( ) function may fail if:

[EBADF]	The catalog descriptor is not valid.
[EINTR]	The *catclose*( ) function was interrupted by a signal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catgets( ), *catopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<nl_types.h>**

CHANGE HISTORY

First released in Issue 2.

NAME

catgets — read a program message

SYNOPSIS

XSI

```
#include <nl_types.h>

char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

DESCRIPTION

The *catgets*( ) function shall attempt to read message *msg_id*, in set *set_id*, from the message catalog identified by *catd*. The *catd* argument is a message catalog descriptor returned from an earlier call to *catopen*( ). The *s* argument points to a default message string which shall be returned by *catgets*( ) if it cannot retrieve the identified message. |

The *catgets*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If the identified message is retrieved successfully, *catgets*( ) shall return a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, *s* shall be returned and *errno* may be set to indicate the error.

ERRORS

The *catgets*( ) function may fail if:

[EBADF]	The *catd* argument is not a valid message catalog descriptor open for reading.
[EBADMSG]	The message identified by *set_id* and *msg_id* in the specified message catalog did not satisfy implementation-defined security criteria.
[EINTR]	The read operation was terminated due to the receipt of a signal, and no data was transferred.
[EINVAL]	The message catalog identified by *catd* is corrupted.
[ENOMSG]	The message identified by *set_id* and *msg_id* is not in the message catalog.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose( ), *catopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<nl_types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 5

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

catopen — open a message catalog

SYNOPSIS

XSI

```
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);
```

DESCRIPTION

The *catopen*( ) function shall open a message catalog and return a message catalog descriptor. The *name* argument specifies the name of the message catalog to be opened. If *name* contains a `'/'`, then *name* specifies a complete name for the message catalog. Otherwise, the environment variable *NLSPATH* is used with *name* substituted for the `%N` conversion specification (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If *NLSPATH* exists in the environment when the process starts, then if the process has appropriate privileges, the behavior of *catopen*( ) is undefined. If *NLSPATH* does not exist in the environment, or if a message catalog cannot be found in any of the components specified by *NLSPATH*, then an implementation-defined default path shall be used. This default may be affected by the setting of *LC_MESSAGES* if the value of *oflag* is NL_CAT_LOCALE, or the *LANG* environment variable if *oflag* is 0. |

A message catalog descriptor shall remain valid in a process until that process closes it, or a successful call to one of the *exec* functions. A change in the setting of the *LC_MESSAGES* category may invalidate existing open catalogs. |

If a file descriptor is used to implement message catalog descriptors, the FD_CLOEXEC flag shall be set; see **<fcntl.h>**.

If the value of the *oflag* argument is 0, the *LANG* environment variable is used to locate the catalog without regard to the *LC_MESSAGES* category. If the *oflag* argument is NL_CAT_LOCALE, the *LC_MESSAGES* category is used to locate the message catalog (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables).

RETURN VALUE

Upon successful completion, *catopen*( ) shall return a message catalog descriptor for use on subsequent calls to *catgets*( ) and *catclose*( ). Otherwise, *catopen*( ) shall return (**nl_catd**) −1 and set *errno* to indicate the error.

ERRORS

The *catopen*( ) function may fail if:

[EACCES]	Search permission is denied for the component of the path prefix of the message catalog or read permission is denied for the message catalog.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENAMETOOLONG]	The length of a pathname of the message catalog exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |
[ENFILE]	Too many files are currently open in the system.
[ENOENT]	The message catalog does not exist or the *name* argument points to an empty string.

[ENOMEM]	Insufficient storage space is available.
[ENOTDIR]	A component of the path prefix of the message catalog is not a directory.

EXAMPLES

None.

APPLICATION USAGE

Some implementations of *catopen*( ) use *malloc*( ) to allocate space for internal buffer areas. The *catopen*( ) function may fail if there is insufficient storage space available to accommodate these buffers.

Conforming applications must assume that message catalog descriptors are not valid after a call to one of the *exec* functions. |

Application writers should be aware that guidelines for the location of message catalogs have not yet been developed. Therefore they should take care to avoid conflicting with catalogs used by other applications and the standard utilities.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catclose( ), *catgets*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<nl_types.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 2.

NAME

cbrt, cbrtf, cbrtl — cube root functions

SYNOPSIS

```
#include <math.h>

double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the real cube root of their argument *x*.

RETURN VALUE

Upon successful completion, these functions shall return the cube root of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

For some applications, a true cube root function, which returns negative results for negative arguments, is more appropriate than *pow*(*x*, 1.0 / 3.0), which returns a NaN for *x* less than 0.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X / OPEN UNIX extension to BASE.

Issue 6

The *cbrt*( ) function is no longer marked as an extension.

The *cbrtf*( ) and *cbrtl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

NAME

ccos, ccosf, ccosl — complex cosine functions

SYNOPSIS

```
#include <complex.h>

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex cosine of *z*.

RETURN VALUE

These functions shall return the complex cosine value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cacos( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

ccosf — complex cosine functions

SYNOPSIS

```
#include <complex.h>

float complex ccosf(float complex z);
```

DESCRIPTION

Refer to *ccos*( ).

NAME

ccosh, ccoshf, ccoshl — complex hyperbolic cosine functions

SYNOPSIS

```
#include <complex.h>

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex hyperbolic cosine of *z*.

RETURN VALUE

These functions shall return the complex hyperbolic cosine value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cacosh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

ccosl — complex cosine functions

SYNOPSIS

```
#include <complex.h>

long double complex ccosl(long double complex z);
```

DESCRIPTION

Refer to *ccos*( ).

NAME

ceil, ceilf, ceill — ceiling value function

SYNOPSIS

```
#include <math.h>

double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the smallest integral value not less than *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, *ceil*( ), *ceilf*( ), and *ceill*( ) shall return the smallest integral value not less than *x*, expressed as a type **double**, **float**, or **long double**, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *ceil*( ), *ceilf*( ), and *ceill*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

XSI Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

The integral value returned by these functions need not be expressible as an **int** or **long**. The return value should be tested before assigning it to an integer type to avoid the undefined results of an integer overflow.

The *ceil*( ) function can only overflow when the floating-point representation has DBL_MANT_DIG > DBL_MAX_EXP.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *floor*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *ceilf*( ) and *ceill*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

cexp, cexpf, cexpl — complex exponential functions

SYNOPSIS

```
#include <complex.h>

double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex exponent of z, defined as e^z.

RETURN VALUE

These functions shall return the complex exponential value of z.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clog( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cfgetispeed — get input baud rate

SYNOPSIS

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termios_p);
```

DESCRIPTION

The *cfgetispeed*( ) function shall extract the input baud rate from the **termios** structure to which the *termios_p* argument points.

This function shall return exactly the value in the **termios** data structure, without interpretation.

RETURN VALUE

Upon successful completion, *cfgetispeed*( ) shall return a value of type **speed_t** representing the input baud rate.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The term *baud* is used historically here, but is not technically correct. This is properly ''bits per second'', which may not be the same as baud. However, the term is used because of the historical usage and understanding.

The *cfgetospeed*( ), *cfgetispeed*( ), *cfsetospeed*( ), and *cfsetispeed*( ) functions do not take arguments as numbers, but rather as symbolic names. There are two reasons for this:

1. Historically, numbers were not used because of the way the rate was stored in the data structure. This is retained even though a function is now used.
2. More importantly, only a limited set of possible rates is at all portable, and this constrains the application to that set.

There is nothing to prevent an implementation to accept, as an extension, a number (such as 126) if it wished, and because the encoding of the Bxxx symbols is not specified, this can be done so no ambiguity is introduced.

Setting the input baud rate to zero was a mechanism to allow for split baud rates. Clarifications in this volume of IEEE Std 1003.1-200x have made it possible to determine whether split rates are supported and to support them without having to treat zero as a special case. Since this functionality is also confusing, it has been declared obsolescent. The 0 argument referred to is the literal constant 0, not the symbolic constant B0. This volume of IEEE Std 1003.1-200x does not preclude B0 from being defined as the value 0; in fact, implementations would likely benefit from the two being equivalent. This volume of IEEE Std 1003.1-200x does not fully specify whether the previous *cfsetispeed*( ) value is retained after a *tcgetattr*( ) as the actual value or as zero. Therefore, conforming applications should always set both the input speed and output speed when setting either. |

In historical implementations, the baud rate information is traditionally kept in **c_cflag**. Applications should be written to presume that this might be the case (and thus not blindly copy **c_cflag**), but not to rely on it in case it is in some other field of the structure. Setting the **c_cflag** field absolutely after setting a baud rate is a non-portable action because of this. In general, the unused parts of the flag fields might be used by the implementation and should not be blindly copied from the descriptions of one terminal device to another.

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetospeed( ), *cfsetispeed*( ), *cfsetospeed*( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

NAME

cfgetospeed — get output baud rate

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(const struct termios *termios_p);
```

DESCRIPTION

The *cfgetospeed*( ) function shall extract the output baud rate from the **termios** structure to which the *termios_p* argument points.

This function shall return exactly the value in the **termios** data structure, without interpretation.

RETURN VALUE

Upon successful completion, *cfgetospeed*( ) shall return a value of type **speed_t** representing the output baud rate.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *cfgetispeed*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed( ), *cfsetispeed*( ), *cfsetospeed*( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

NAME

cfsetispeed — set input baud rate

SYNOPSIS

```
#include <termios.h>

int cfsetispeed(struct termios *termios_p, speed_t speed);
```

DESCRIPTION

The *cfsetispeed*( ) function shall set the input baud rate stored in the structure pointed to by *termios_p* to *speed*.

There shall be no effect on the baud rates set in the hardware until a subsequent successful call to *tcsetattr*( ) with the same **termios** structure. Similarly, errors resulting from attempts to set baud rates not supported by the terminal device need not be detected until the *tcsetattr*( ) function is called.

RETURN VALUE

Upon successful completion, *cfsetispeed*( ) shall return 0; otherwise, −1 shall be returned, and *errno* may be set to indicate the error.

ERRORS

The *cfsetispeed*( ) function may fail if:

[EINVAL]	The *speed* value is not a valid baud rate.
[EINVAL]	The value of *speed* is outside the range of possible speed values as specified in **<termios.h>**.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *cfgetispeed*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed( ), *cfgetospeed*( ), *cfsetospeed*( ), *tcsetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

NAME

cfsetospeed — set output baud rate

SYNOPSIS

```
#include <termios.h>

int cfsetospeed(struct termios *termios_p, speed_t speed);
```

DESCRIPTION

The *cfsetospeed*( ) function shall set the output baud rate stored in the structure pointed to by *termios_p* to *speed*.

There shall be no effect on the baud rates set in the hardware until a subsequent successful call to *tcsetattr*( ) with the same **termios** structure. Similarly, errors resulting from attempts to set baud rates not supported by the terminal device need not be detected until the *tcsetattr*( ) function is called.

RETURN VALUE

Upon successful completion, *cfsetospeed*( ) shall return 0; otherwise, it shall return −1 and *errno* may be set to indicate the error.

ERRORS

The *cfsetospeed*( ) function may fail if:

[EINVAL]	The *speed* value is not a valid baud rate.
[EINVAL]	The value of *speed* is outside the range of possible speed values as specified in **<termios.h>**.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *cfgetispeed*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

cfgetispeed( ), *cfgetospeed*( ), *cfsetispeed*( ), *tcsetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* and the [EINVAL] error conditions are added.

NAME

chdir — change working directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *path);
```

DESCRIPTION

The *chdir*( ) function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with `'/'`.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned, the current working directory shall remain unchanged, and *errno* shall be set to indicate the error.

ERRORS

The *chdir*( ) function shall fail if:

[EACCES]	Search permission is denied for any component of the pathname.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing directory or *path* is an empty string.
[ENOTDIR]	A component of the pathname is not a directory.

The *chdir*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Changing the Current Working Directory

The following example makes the value pointed to by **directory**, **/tmp**, the current working directory.

```
#include <unistd.h>
...
char *directory = "/tmp";
int ret;

ret = chdir (directory);
```

APPLICATION USAGE

None.

RATIONALE

The *chdir*( ) function only affects the working directory of the current process.

FUTURE DIRECTIONS

None.

SEE ALSO

getcwd( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The APPLICATION USAGE section is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

chmod — change mode of a file

SYNOPSIS

```
#include <sys/stat.h>

int chmod(const char *path, mode_t mode);
```

DESCRIPTION

XSI The *chmod*( ) function shall change S_ISUID, S_ISGID, S_ISVTX, and the file permission bits of the file named by the pathname pointed to by the *path* argument to the corresponding bits in the *mode* argument. The application shall ensure that the effective user ID of the process matches the owner of the file or the process has appropriate privileges in order to do this.

XSI S_ISUID, S_ISGID, S_ISVTX, and the file permission bits are described in **<sys/stat.h>**.

If the calling process does not have appropriate privileges, and if the group ID of the file does not match the effective group ID or one of the supplementary group IDs and if the file is a regular file, bit S_ISGID (set-group-ID on execution) in the file's mode shall be cleared upon successful return from *chmod*( ).

Additional implementation-defined restrictions may cause the S_ISUID and S_ISGID bits in *mode* to be ignored.

The effect on file descriptors for files open at the time of a call to *chmod*( ) is implementation-defined.

Upon successful completion, *chmod*( ) shall mark for update the *st_ctime* field of the file.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error. If −1 is returned, no change to the file mode occurs.

ERRORS

The *chmod*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[EPERM]	The effective user ID does not match the owner of the file and the process does not have appropriate privileges.
[EROFS]	The named file resides on a read-only file system.

The *chmod*( ) function may fail if:

[EINTR]	A signal was caught during execution of the function.
[EINVAL]	The value of the *mode* argument is invalid.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname strings exceeded {PATH_MAX}.

EXAMPLES

Setting Read Permissions for User, Group, and Others

The following example sets read permissions for the owner, group, and others.

```
#include <sys/stat.h>

const char *path;
...
chmod(path, S_IRUSR|S_IRGRP|S_IROTH);
```

Setting Read, Write, and Execute Permissions for the Owner Only

The following example sets read, write, and execute permissions for the owner, and no permissions for group and others.

```
#include <sys/stat.h>

const char *path;
...
chmod(path, S_IRWXU);
```

Setting Different Permissions for Owner, Group, and Other

The following example sets owner permissions for CHANGEFILE to read, write, and execute, group permissions to read and execute, and other permissions to read.

```
#include <sys/stat.h>

#define CHANGEFILE "/etc/myfile"
...
chmod(CHANGEFILE, S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH);
```

Setting and Checking File Permissions

The following example sets the file permission bits for a file named **/home/cnd/mod1**, then calls the *stat*( ) function to verify the permissions.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
struct stat buffer
...
chmod("home/cnd/mod1", S_IRWXU|S_IRWXG|S_IROTH|S_IWOTH);
status = stat("home/cnd/mod1", &buffer;);
```

APPLICATION USAGE

In order to ensure that the S_ISUID and S_ISGID bits are set, an application requiring this should use *stat*( ) after a successful *chmod*( ) to verify this.

Any file descriptors currently open by any process on the file could possibly become invalid if the mode of the file is changed to a value which would deny access to that process. One situation where this could occur is on a stateless file system. This behavior will not occur in a conforming environment.

RATIONALE

This volume of IEEE Std 1003.1-200x specifies that the S_ISGID bit is cleared by *chmod*( ) on a regular file under certain conditions. This is specified on the assumption that regular files may be executed, and the system should prevent users from making executable *setgid*( ) files perform with privileges that the caller does not have. On implementations that support execution of other file types, the S_ISGID bit should be cleared for those file types under the same circumstances.

Implementations that use the S_ISUID bit to indicate some other function (for example, mandatory record locking) on non-executable files need not clear this bit on writing. They should clear the bit for executable files and any other cases where the bit grants special powers to processes that change the file contents. Similar comments apply to the S_ISGID bit.

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), *mkdir*( ), *mkfifo*( ), *open*( ), *stat*( ), *statvfs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EINVAL] and [EINTR] optional error conditions are added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

chown — change owner and group of a file

SYNOPSIS

```
#include <unistd.h>

int chown(const char *path, uid_t owner, gid_t group);
```

DESCRIPTION

The *chown*( ) function shall change the user and group ownership of a file.

The *path* argument points to a pathname naming a file. The user ID and group ID of the named file shall be set to the numeric values contained in *owner* and *group*, respectively.

Only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file. If _POSIX_CHOWN_RESTRICTED is in effect for *path*:

- Changing the user ID is restricted to processes with appropriate privileges.
- Changing the group ID is permitted to a process with an effective user ID equal to the user ID of the file, but without appropriate privileges, if and only if *owner* is equal to the file's user ID or (**uid_t**)–1 and *group* is equal either to the calling process' effective group ID or to one of its supplementary group IDs.

If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, and the process does not have appropriate privileges, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful return from *chown*( ). If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, and the process has appropriate privileges, it is implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the *chown*( ) function is successfully invoked on a file that is not a regular file and one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, the set-user-ID and set-group-ID bits may be cleared.

If *owner* or *group* is specified as (**uid_t**)–1 or (**gid_t**)–1, respectively, the corresponding ID of the file shall not be changed. If both owner and group are –1, the times need not be updated.

Upon successful completion, *chown*( ) shall mark for update the *st_ctime* field of the file.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned and *errno* set to indicate the error. If –1 is returned, no changes are made in the user ID and group ID of the file.

ERRORS

The *chown*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.

[EPERM]	The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.
[EROFS]	The named file resides on a read-only file system.

The *chown*( ) function may fail if:

[EIO]	An I/O error occurred while reading or writing to the file system.
[EINTR]	The *chown*( ) function was interrupted by a signal which was caught.
[EINVAL]	The owner or group ID supplied is not a value supported by the implementation.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

Although *chown*( ) can be used on some implementations by the file owner to change the owner and group to any desired values, the only portable use of this function is to change the group of a file to the effective GID of the calling process or to a member of its group set.

RATIONALE

System III and System V allow a user to give away files; that is, the owner of a file may change its user ID to anything. This is a serious problem for implementations that are intended to meet government security regulations. Version 7 and 4.3 BSD permit only the superuser to change the user ID of a file. Some government agencies (usually not ones concerned directly with security) find this limitation too confining. This volume of IEEE Std 1003.1-200x uses *may* to permit secure implementations while not disallowing System V.

System III and System V allow the owner of a file to change the group ID to anything. Version 7 permits only the superuser to change the group ID of a file. 4.3 BSD permits the owner to change the group ID of a file to its effective group ID or to any of the groups in the list of supplementary group IDs, but to no others.

The POSIX.1-1990 standard requires that the *chown*( ) function invoked by a non-appropriate privileged process clear the S_ISGID and the S_ISUID bits for regular files, and permits them to be cleared for other types of files. This is so that changes in accessibility do not accidentally cause files to become security holes. Unfortunately, requiring these bits to be cleared on non-executable data files also clears the mandatory file locking bit (shared with S_ISGID), which is an extension on many implementations (it first appeared in System V). These bits should only be required to be cleared on regular files that have one or more of their execute bits set.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *pathconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The wording describing the optional dependency on _POSIX_CHOWN_RESTRICTED is restored.
- The [EPERM] error is restored as an error dependent on _POSIX_CHOWN_RESTRICTED. This is since its operand is a pathname and applications should be aware that the error may not occur for that pathname if the file system does not support _POSIX_CHOWN_RESTRICTED.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The value for *owner* of (**uid_t**)–1 allows the use of –1 by the owner of a file to change the group ID only. A corresponding change is made for group.
- The [ELOOP] mandatory error condition is added.
- The [EIO] and [EINTR] optional error conditions are added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that the S_ISUID and S_ISGID bits do not need to be cleared when the process has appropriate privileges.
- The [ELOOP] optional error condition is added.

NAME

cimag, cimagf, cimagl — complex imaginary functions

SYNOPSIS

```
#include <complex.h>

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the imaginary part of *z*.

RETURN VALUE

These functions shall return the imaginary part value (as a real).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *conj*( ), *cproj*( ), *creal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

clearerr — clear indicators on a stream

SYNOPSIS

```
#include <stdio.h>

void clearerr(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *clearerr*( ) function shall clear the end-of-file and error indicators for the stream to which *stream* points.

RETURN VALUE

The *clearerr*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

clock — report CPU time used

SYNOPSIS

```
#include <time.h>

clock_t clock(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *clock*( ) function shall return the implementation's best approximation to the processor time used by the process since the beginning of an implementation-defined era related only to the process invocation.

RETURN VALUE

To determine the time in seconds, the value returned by *clock*( ) should be divided by the value of the macro CLOCKS_PER_SEC. XSI CLOCKS_PER_SEC is defined to be one million in **<time.h>**. If the processor time used is not available or its value cannot be represented, the function shall return the value (**clock_t**)–1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In order to measure the time spent in a program, *clock*( ) should be called at the start of the program and its return value subtracted from the value returned by subsequent calls. The value returned by *clock*( ) is defined for compatibility across systems that have clocks with different resolutions. The resolution on any particular system need not be to microsecond accuracy.

The value returned by *clock*( ) may wrap around on some implementations. For example, on a machine with 32-bit values for **clock_t**, it wraps after 2 147 seconds or 36 minutes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

clock_getcpuclockid — access a process CPU-time clock (**ADVANCED REALTIME**)

SYNOPSIS

CPT
```
#include <time.h>

int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

DESCRIPTION

The *clock_getcpuclockid*( ) function shall return the clock ID of the CPU-time clock of the process specified by *pid*. If the process described by *pid* exists and the calling process has permission, the clock ID of this clock shall be returned in *clock_id*.

If *pid* is zero, the *clock_getcpuclockid*( ) function shall return the clock ID of the CPU-time clock of the process making the call, in *clock_id*.

The conditions under which one process has permission to obtain the CPU-time clock ID of other processes are implementation-defined.

RETURN VALUE

Upon successful completion, *clock_getcpuclockid*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *clock_getcpuclockid*( ) function shall fail if:

[EPERM]	The requesting process does not have permission to access the CPU-time clock for the process.

The *clock_getcpuclockid*( ) function may fail if:

[ESRCH]	No process can be found corresponding to the process specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

The *clock_getcpuclockid*( ) function is part of the Process CPU-Time Clocks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *timer_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

clock_getres, clock_gettime, clock_settime — clock and timer functions (**REALTIME**)

SYNOPSIS

TMR

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

DESCRIPTION

The *clock_getres*( ) function shall return the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the argument *res* is not NULL, the resolution of the specified clock shall be stored in the location pointed to by *res*. If *res* is NULL, the clock resolution is not returned. If the *time* argument of *clock_settime*( ) is not a multiple of *res*, then the value is truncated to a multiple of *res*.

The *clock_gettime*( ) function shall return the current value *tp* for the specified clock, *clock_id*.

The *clock_settime*( ) function shall set the specified clock, *clock_id*, to the value specified by *tp*. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock shall be truncated down to the smaller multiple of the resolution.

A clock may be system-wide (that is, visible to all processes) or per-process (measuring time that is meaningful only within a process). All implementations shall support a *clock_id* of CLOCK_REALTIME as defined in **<time.h>**. This clock represents the realtime clock for the system. For this clock, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation may also support additional clocks. The interpretation of time values for these clocks is unspecified.

If the value of the CLOCK_REALTIME clock is set via *clock_settime*( ), the new value of the clock shall be used to determine the time of expiration for absolute time services based upon the CLOCK_REALTIME clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service shall expire immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock via *clock_settime*( ) shall have no effect on threads that are blocked waiting for a relative time service based upon this clock, including the *nanosleep*( ) function; nor on the expiration of relative timers based upon this clock. Consequently, these time services shall expire when the requested relative interval elapses, independently of the new or old value of the clock.

MON If the Monotonic Clock option is supported, all implementations shall support a *clock_id* of CLOCK_MONOTONIC defined in **<time.h>**. This clock represents the monotonic clock for the system. For this clock, the value returned by *clock_gettime*( ) represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK_MONOTONIC clock cannot be set via *clock_settime*( ). This function shall fail if it is invoked with a *clock_id* argument of CLOCK_MONOTONIC.

The effect of setting a clock via *clock_settime*( ) on armed per-process timers associated with a clock other than CLOCK_REALTIME is implementation-defined.

CS If the value of the CLOCK_REALTIME clock is set via *clock_settime*( ), the new value of the clock shall be used to determine the time at which the system shall awaken a thread blocked on an absolute *clock_nanosleep*( ) call based upon the CLOCK_REALTIME clock. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the call shall return immediately as if the clock had reached the requested time normally.

Setting the value of the CLOCK_REALTIME clock via *clock_settime*( ) shall have no effect on any thread that is blocked on a relative *clock_nanosleep*( ) call. Consequently, the call shall return when the requested relative interval elapses, independently of the new or old value of the clock.

The appropriate privilege to set a particular clock is implementation-defined.

CPT If _POSIX_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *clock_getcpuclockid*( ), which represent the CPU-time clock of a given process. Implementations shall also support the special **clockid_t** value CLOCK_PROCESS_CPUTIME_ID, which represents the CPU-time clock of the calling process when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy (see **Scheduling Policies** (on page 494)).

TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support clock ID values obtained by invoking *pthread_getcpuclockid*( ), which represent the CPU-time clock of a given thread. Implementations shall also support the special **clockid_t** value CLOCK_THREAD_CPUTIME_ID, which represents the CPU-time clock of the calling thread when invoking one of the *clock_**( ) or *timer_**( ) functions. For these clock IDs, the values returned by *clock_gettime*( ) and specified by *clock_settime*( ) shall represent the amount of | execution time of the thread associated with the clock. Changing the value of a CPU-time clock | via *clock_settime*( ) shall have no effect on the behavior of the sporadic server scheduling policy | (see **Scheduling Policies** (on page 494)).

RETURN VALUE

A return value of 0 shall indicate that the call succeeded. A return value of −1 shall indicate that an error occurred, and *errno* shall be set to indicate the error.

ERRORS

The *clock_getres*( ), *clock_gettime*( ), and *clock_settime*( ) functions shall fail if:

[EINVAL] The *clock_id* argument does not specify a known clock.

The *clock_settime*( ) function shall fail if:

[EINVAL] The *tp* argument to *clock_settime*( ) is outside the range for the given clock ID.

[EINVAL] The *tp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million.

MON [EINVAL] The value of the *clock_id* argument is CLOCK_MONOTONIC.

The *clock_settime*( ) function may fail if:

[EPERM] The requesting process does not have the appropriate privilege to set the specified clock.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Timers option and need not be available on all implementations.

Note that the absolute value of the monotonic clock is meaningless (because its origin is arbitrary), and thus there is no need to set it. Furthermore, realtime applications can rely on the fact that the value of this clock is never set and, therefore, that time intervals measured with this clock will not be affected by calls to *clock_settime*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getcpuclockid( ), *clock_nanosleep*( ), *ctime*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *nanosleep*( ), *pthread_mutex_timedlock*( ), *sem_timedwait*( ), *time*( ), *timer_create*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The APPLICATION USAGE section is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added of the effect of resetting the clock resolution.

CPU-time clocks and the *clock_getcpuclockid*( ) function are added for alignment with IEEE Std 1003.1d-1999.

The following changes are added for alignment with IEEE Std 1003.1j-2000:

- The DESCRIPTION is updated as follows:
  - The value returned by *clock_gettime*( ) for CLOCK_MONOTONIC is specified.
  - *clock_settime*( ) failing for CLOCK_MONOTONIC is specified.
  - The effects of *clock_settime*( ) on the *clock_nanosleep*( ) function with respect to CLOCK_REALTIME is specified.
- An [EINVAL] error is added to the ERRORS section, indicating that *clock_settime*( ) fails for CLOCK_MONOTONIC.
- The APPLICATION USAGE section notes that the CLOCK_MONOTONIC clock need not and shall not be set by *clock_settime*( ) since the absolute value of the CLOCK_MONOTONIC clock is meaningless.
- The *clock_nanosleep*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *pthread_mutex_timedlock*( ), *sem_timedwait*( ), *timer_create*( ), and *timer_settime*( ) functions are added to the SEE ALSO section.

NAME

clock_nanosleep — high resolution sleep with specifiable clock (**ADVANCED REALTIME**)

SYNOPSIS

CS

```
#include <time.h>
```

```
int clock_nanosleep(clockid_t clock_id, int flags,
    const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

If the flag TIMER_ABSTIME is not set in the *flags* argument, the *clock_nanosleep*( ) function shall cause the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The clock used to measure the time shall be the clock specified by *clock_id*.

If the flag TIMER_ABSTIME is set in the *flags* argument, the *clock_nanosleep*( ) function shall cause the current thread to be suspended from execution until either the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then *clock_nanosleep*( ) shall return immediately and the calling process shall not be suspended.

The suspension time caused by this function may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time for the relative *clock_nanosleep*( ) function (that is, with the TIMER_ABSTIME flag not set) shall not be less than the time interval specified by *rqtp*, as measured by the corresponding clock. The suspension for the absolute *clock_nanosleep*( ) function (that is, with the TIMER_ABSTIME flag set) shall be in effect at least until the value of the corresponding clock reaches the absolute time specified by *rqtp*, except for the case of being interrupted by a signal.

The use of the *clock_nanosleep*( ) function shall have no effect on the action or blockage of any signal.

The *clock_nanosleep*( ) function shall fail if the *clock_id* argument refers to the CPU-time clock of the calling thread. It is unspecified if *clock_id* values of other CPU-time clocks are allowed.

RETURN VALUE

If the *clock_nanosleep*( ) function returns because the requested time has elapsed, its return value shall be zero.

If the *clock_nanosleep*( ) function returns because it has been interrupted by a signal, it shall return the corresponding error value. For the relative *clock_nanosleep*( ) function, if the *rmtp* argument is non-NULL, the **timespec** structure referenced by it shall be updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned. The absolute *clock_nanosleep*( ) function has no effect on the structure referenced by *rmtp*.

If *clock_nanosleep*( ) fails, it shall return the corresponding error value.

ERRORS

The *clock_nanosleep*( ) function shall fail if:

[EINTR]	The *clock_nanosleep*( ) function was interrupted by a signal.
[EINVAL]	The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1 000 million; or the TIMER_ABSTIME flag was specified in flags and the *rqtp* argument is outside the range for the clock specified by *clock_id*; or the *clock_id* argument does not specify a known clock, or specifies the CPU-time clock of the calling thread.
[ENOTSUP]	The *clock_id* argument specifies a clock for which *clock_nanosleep*( ) is not supported, such as a CPU-time clock.

EXAMPLES

None.

APPLICATION USAGE

Calling *clock_nanosleep*( ) with the value TIMER_ABSTIME not set in the *flags* argument and with a *clock_id* of CLOCK_REALTIME is equivalent to calling *nanosleep*( ) with the same *rqtp* and *rmtp* arguments.

RATIONALE

The *nanosleep*( ) function specifies that the system-wide clock CLOCK_REALTIME is used to measure the elapsed time for this time service. However, with the introduction of the monotonic clock CLOCK_MONOTONIC a new relative sleep function is needed to allow an application to take advantage of the special characteristics of this clock.

There are many applications in which a process needs to be suspended and then activated multiple times in a periodic way; for example, to poll the status of a non-interrupting device or to refresh a display device. For these cases, it is known that precise periodic activation cannot be achieved with a relative *sleep*( ) or *nanosleep*( ) function call. Suppose, for example, a periodic process that is activated at time $T0$, executes for a while, and then wants to suspend itself until time $T0+T$, the period being T. If this process wants to use the *nanosleep*( ) function, it must first call *clock_gettime*( ) to get the current time, then calculate the difference between the current time and $T0+T$ and, finally, call *nanosleep*( ) using the computed interval. However, the process could be preempted by a different process between the two function calls, and in this case the interval computed would be wrong; the process would wake up later than desired. This problem would not occur with the absolute *clock_nanosleep*( ) function, since only one function call would be necessary to suspend the process until the desired time. In other cases, however, a relative sleep is needed, and that is why both functionalities are required.

Although it is possible to implement periodic processes using the timers interface, this implementation would require the use of signals, and the reservation of some signal numbers. In this regard, the reasons for including an absolute version of the *clock_nanosleep*( ) function in IEEE Std 1003.1-200x are the same as for the inclusion of the relative *nanosleep*( ).

It is also possible to implement precise periodic processes using *pthread_cond_timedwait*( ), in which an absolute timeout is specified that takes effect if the condition variable involved is never signaled. However, the use of this interface is unnatural, and involves performing other operations on mutexes and condition variables that imply an unnecessary overhead. Furthermore, *pthread_cond_timedwait*( ) is not available in implementations that do not support threads.

Although the interface of the relative and absolute versions of the new high resolution sleep service is the same *clock_nanosleep*( ) function, the *rmtp* argument is only used in the relative sleep. This argument is needed in the relative *clock_nanosleep*( ) function to reissue the function call if it is interrupted by a signal, but it is not needed in the absolute *clock_nanosleep*( ) function call; if the call is interrupted by a signal, the absolute *clock_nanosleep*( ) function can be invoked again with the same *rqtp* argument used in the interrupted call.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *nanosleep*( ), *pthread_cond_timedwait*( ), *sleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

clock_settime — clock and timer functions (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

DESCRIPTION

Refer to *clock_getres*( ).

NAME

clog, clogf, clogl — complex natural logarithm functions

SYNOPSIS

```
#include <complex.h>

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex natural (base *e*) logarithm of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cexp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

close — close a file descriptor

SYNOPSIS

```
#include <unistd.h>

int close(int fildes);
```

DESCRIPTION

The *close*( ) function shall deallocate the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to *open*( ) or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked).

If *close*( ) is interrupted by a signal that is to be caught, it shall return −1 with *errno* set to [EINTR] and the state of *fildes* is unspecified. If an I/O error occurred while reading from or writing to the file system during *close*( ), it may return −1 with *errno* set to [EIO]; if this error is returned, the state of *fildes* is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO shall be discarded.

When all file descriptors associated with an open file description have been closed the open file description shall be freed.

If the link count of the file is 0, when all file descriptors associated with the file are closed, the space occupied by the file shall be freed and the file shall no longer be accessible.

XSR If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive a SIGPOLL signal for events associated with that STREAM, the calling process shall be unregistered for events associated with the STREAM. The last *close*( ) for a STREAM shall cause the STREAM associated with *fildes* to be dismantled. If O_NONBLOCK is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, *close*( ) shall wait for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed via an I_SETCLTIME *ioctl*( ) request. If the O_NONBLOCK flag is set, or if there are any pending signals, *close*( ) shall not wait for output to drain, and shall dismantle the STREAM immediately.

If the implementation supports STREAMS-based pipes, and *fildes* is associated with one end of a pipe, the last *close*( ) shall cause a hangup to occur on the other end of the pipe. In addition, if the other end of the pipe has been named by *fattach*( ), then the last *close*( ) shall force the named end to be detached by *fdetach*( ). If the named end has no open file descriptors associated with it and gets detached, the STREAM associated with that end shall also be dismantled.

XSI If *fildes* refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal shall be sent to the process group, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

XSR If *fildes* refers to the slave side of a STREAMS-based pseudo-terminal, a zero-length message may be sent to the master.

AIO When there is an outstanding cancelable asynchronous I/O operation against *fildes* when *close*( ) is called, that I/O operation may be canceled. An I/O operation that is not canceled completes as if the *close*( ) operation had not yet occurred. All operations that are not canceled shall complete as if the *close*( ) blocked until the operations completed. The *close*( ) operation itself need not block awaiting such I/O completion. Whether any I/O operation is canceled, and which I/O operation may be canceled upon *close*( ), is implementation-defined.

MF|SHM If a shared memory object or a memory mapped file remains referenced at the last close (that is, a process has it mapped), then the entire contents of the memory object shall persist until the memory object becomes unreferenced. If this is the last close of a shared memory object or a memory mapped file and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, then the memory object shall be removed.

If *fildes* refers to a socket, *close*( ) shall cause the socket to be destroyed. If the socket is in connection-mode, and the SO_LINGER option is set for the socket with non-zero linger time, and the socket has untransmitted data, then *close*( ) shall block for up to the current linger interval until all data is transmitted.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *close*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINTR]	The *close*( ) function was interrupted by a signal.

The *close*( ) function may fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.

EXAMPLES

Reassigning a File Descriptor

The following example closes the file descriptor associated with standard output for the current process, re-assigns standard output to a new file descriptor, and closes the original file descriptor to clean up. This example assumes that the file descriptor 0 (which is the descriptor for standard input) is not closed.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Incidentally, this is exactly what could be achieved using:

```
dup2(pfd, 1);
close(pfd);
```

Closing a File Descriptor

In the following example, *close*( ) is used to close a file descriptor after an unsuccessful attempt is made to associate that file descriptor with a stream.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#define LOCKFILE "/etc/ptmp"
...
int pfd;
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
...
```

APPLICATION USAGE

An application that had used the *stdio* routine *fopen*( ) to open a file should use the corresponding *fclose*( ) routine rather than *close*( ). Once a file is closed, the file descriptor no longer exists, since the integer corresponding to it no longer refers to a file.

RATIONALE

The use of interruptible device close routines should be discouraged to avoid problems with the implicit closes of file descriptors by *exec* and *exit*( ). This volume of IEEE Std 1003.1-200x only intends to permit such behavior by specifying the [EINTR] error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

fattach( ), *fclose*( ), *fdetach*( ), *fopen*( ), *ioctl*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Issue 6

The DESCRIPTION related to a STREAMS-based file or pseudo-terminal is marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error condition is added as an optional error.
- The DESCRIPTION is updated to describe the state of the *fildes* file descriptor as unspecified if an I/O error occurs and an [EIO] error condition is returned.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory objects and memory mapped files (and not typed memory objects) are the types of memory objects to which the paragraph on last closes applies.

NAME

closedir — close a directory stream

SYNOPSIS

```
#include <dirent.h>

int closedir(DIR *dirp);
```

DESCRIPTION

The *closedir*( ) function shall close the directory stream referred to by the argument *dirp*. Upon return, the value of *dirp* may no longer point to an accessible object of the type **DIR**. If a file descriptor is used to implement type **DIR**, that file descriptor shall be closed.

RETURN VALUE

Upon successful completion, *closedir*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *closedir*( ) function may fail if:

[EBADF]	The *dirp* argument does not refer to an open directory stream.
[EINTR]	The *closedir*( ) function was interrupted by a signal.

EXAMPLES

Closing a Directory Stream

The following program fragment demonstrates how the *closedir*( ) function is used.

```
...
    DIR *dir;
    struct dirent *dp;
...
    if ((dir = opendir (".")) == NULL) {
...
    }

    while ((dp = readdir (dir)) != NULL) {
...
    }

    closedir(dir);
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

opendir( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EINTR] error condition is added as an optional error condition.

NAME

closelog, openlog, setlogmask, syslog — control system log

SYNOPSIS

XSI
```
#include <syslog.h>

void closelog(void);
void openlog(const char *ident, int logopt, int facility);
int setlogmask(int maskpri);
void syslog(int priority, const char *message, ... /* arguments */);
```

DESCRIPTION

The *syslog*( ) function shall send a message to an implementation-defined logging facility, which may log it in an implementation-defined system log, write it to the system console, forward it to a list of users, or forward it to the logging facility on another host over the network. The logged message shall include a message header and a message body. The message header contains at least a timestamp and a tag string.

The message body is generated from the *message* and following arguments in the same manner as if these were arguments to *printf*( ), except that the additional conversion specification `%m` shall be recognized; it shall convert no arguments, shall cause the output of the error message string associated with the value of *errno* on entry to *syslog*( ), and may be mixed with argument specifications of the `"%n$"` form. If a complete conversion specification with the `m` conversion specifier character is not just `%m`, the behavior is undefined. A trailing <newline> may be added if needed.

Values of the *priority* argument are formed by OR'ing together a severity level value and an optional facility value. If no facility value is specified, the current default facility value is used.

Possible values of severity level include:

LOG_EMERG	A panic condition.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but that may require special handling.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

The facility indicates the application or system component generating the message. Possible facility values include:

LOG_USER	Messages generated by arbitrary processes. This is the default facility identifier if none is specified.
LOG_LOCAL0	Reserved for local use.

LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.
LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

The *openlog*( ) function shall set process attributes that affect subsequent calls to *syslog*( ). The *ident* argument is a string that is prepended to every message. The *logopt* argument indicates logging options. Values for *logopt* are constructed by a bitwise-inclusive OR of zero or more of the following:

LOG_PID	Log the process ID with each message. This is useful for identifying specific processes.
LOG_CONS	Write messages to the system console if they cannot be sent to the logging facility. The *syslog*( ) function ensures that the process does not acquire the console as a controlling terminal in the process of writing the message.
LOG_NDELAY	Open the connection to the logging facility immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated.
LOG_ODELAY	Delay open until *syslog*( ) is called.
LOG_NOWAIT	Do not wait for child processes that may have been created during the course of logging the message. This option should be used by processes that enable notification of child termination using SIGCHLD, since *syslog*( ) may otherwise block waiting for a child whose exit status has already been collected.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is LOG_USER.

The *openlog*( ) and *syslog*( ) functions may allocate a file descriptor. It is not necessary to call *openlog*( ) prior to calling *syslog*( ).

The *closelog*( ) function shall close any open file descriptors allocated by previous calls to *openlog*( ) or *syslog*( ).

The *setlogmask*( ) function shall set the log priority mask for the current process to *maskpri* and return the previous mask. If the *maskpri* argument is 0, the current log mask is not modified. Calls by the current process to *syslog*( ) with a priority not set in *maskpri* shall be rejected. The default log mask allows all priorities to be logged. A call to *openlog*( ) is not required prior to calling *setlogmask*( ).

Symbolic constants for use as values of the *logopt*, *facility*, *priority*, and *maskpri* arguments are defined in the **<syslog.h>** header.

RETURN VALUE

The *setlogmask*( ) function shall return the previous log priority mask. The *closelog*( ), *openlog*( ), and *syslog*( ) functions shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Using openlog( )

The following example causes subsequent calls to *syslog*( ) to log the process ID with each message, and to write messages to the system console if they cannot be sent to the logging facility.

```
#include <syslog.h>

char *ident = "Process demo";
int logopt = LOG_PID | LOG_CONS;
int facility = LOG_USER;
...
openlog(ident, logopt, facility);
```

Using setlogmask( )

The following example causes subsequent calls to *syslog*( ) to accept error messages or messages generated by arbitrary processes, and to reject all other messages.

```
#include <syslog.h>

int result;
int mask = LOG_MASK (LOG_ERR | LOG_USER);
...
result = setlogmask(mask);
```

Using syslog

The following example sends the message `"This is a message"` to the default logging facility, marking the message as an error message generated by random processes.

```
#include <syslog.h>

char *message = "This is a message";
int priority = LOG_ERR | LOG_USER;
...
syslog(priority, message);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

printf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<syslog.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

confstr — get configurable variables

SYNOPSIS

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

DESCRIPTION

The *confstr*( ) function shall return configuration-defined string values. Its use and purpose are similar to *sysconf*( ), but it is used where string values rather than numeric values are returned. |

The *name* argument represents the system variable to be queried. The implementation shall support the following name values, defined in **<unistd.h>**. It may support others:

_CS_PATH
_CS_POSIX_V6_ILP32_OFF32_CFLAGS
_CS_POSIX_V6_ILP32_OFF32_LDFLAGS
_CS_POSIX_V6_ILP32_OFF32_LIBS
_CS_POSIX_V6_ILP32_OFF32_LINTFLAGS
_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
_CS_POSIX_V6_ILP32_OFFBIG_LIBS
_CS_POSIX_V6_ILP32_OFFBIG_LINTFLAGS
_CS_POSIX_V6_LP64_OFF64_CFLAGS
_CS_POSIX_V6_LP64_OFF64_LDFLAGS
_CS_POSIX_V6_LP64_OFF64_LIBS
_CS_POSIX_V6_LP64_OFF64_LINTFLAGS
_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
_CS_POSIX_V6_LPBIG_OFFBIG_LIBS
_CS_POSIX_V6_LPBIG_OFFBIG_LINTFLAGS
XSI _CS_XBS5_ILP32_OFF32_CFLAGS (**LEGACY**)
_CS_XBS5_ILP32_OFF32_LDFLAGS (**LEGACY**)
_CS_XBS5_ILP32_OFF32_LIBS (**LEGACY**)
_CS_XBS5_ILP32_OFF32_LINTFLAGS (**LEGACY**)
_CS_XBS5_ILP32_OFFBIG_CFLAGS (**LEGACY**)
_CS_XBS5_ILP32_OFFBIG_LDFLAGS (**LEGACY**)
_CS_XBS5_ILP32_OFFBIG_LIBS (**LEGACY**)
_CS_XBS5_ILP32_OFFBIG_LINTFLAGS (**LEGACY**)
_CS_XBS5_LP64_OFF64_CFLAGS (**LEGACY**)
_CS_XBS5_LP64_OFF64_LDFLAGS (**LEGACY**)
_CS_XBS5_LP64_OFF64_LIBS (**LEGACY**)
_CS_XBS5_LP64_OFF64_LINTFLAGS (**LEGACY**)
_CS_XBS5_LPBIG_OFFBIG_CFLAGS (**LEGACY**)
_CS_XBS5_LPBIG_OFFBIG_LDFLAGS (**LEGACY**)
_CS_XBS5_LPBIG_OFFBIG_LIBS (**LEGACY**)
_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS (**LEGACY**)

If *len* is not 0, and if *name* has a configuration-defined value, *confstr*( ) shall copy that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null, then *confstr*( ) shall truncate the string to *len*−1 bytes and null-terminate the result. The application can detect that the string was truncated by comparing the value returned by *confstr*( ) with *len*.

If *len* is 0 and *buf* is a null pointer, then *confstr*( ) shall still return the integer value as defined below, but shall not return a string. If *len* is 0 but *buf* is not a null pointer, the result is unspecified.

If the implementation supports the Shell option, the string stored in *buf* after a call to:

```
confstr(_CS_PATH, buf, sizeof(buf))
```

can be used as a value of the *PATH* environment variable that accesses all of the standard utilities of IEEE Std 1003.1-200x, if the return value is less than or equal to *sizeof*(*buf*).

RETURN VALUE

If *name* has a configuration-defined value, *confstr*( ) shall return the size of buffer that would be needed to hold the entire configuration-defined value including the terminating null. If this return value is greater than *len*, the string returned in *buf* is truncated.

If *name* is invalid, *confstr*( ) shall return 0 and set *errno* to indicate the error.

If *name* does not have a configuration-defined value, *confstr*( ) shall return 0 and leave *errno* unchanged.

ERRORS

The *confstr*( ) function shall fail if:

[EINVAL]	The value of the *name* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

An application can distinguish between an invalid *name* parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if *errno* is modified. This mirrors the behavior of *sysconf*( ).

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable *PATH*. Since *PATH* can be modified by the user to include directories that could contain utilities replacing the standard utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x, applications need a way to determine the system-supplied *PATH* environment variable value that contains the correct search path for the standard utilities.

An application could use:

```
confstr(name, (char *)NULL, (size_t)0)
```

to find out how big a buffer is needed for the string value; use *malloc*( ) to allocate a buffer to hold the string; and call *confstr*( ) again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (perhaps 512 or 1 024 bytes), but then use *malloc*( ) to allocate a larger buffer if it finds that this is too small.

RATIONALE

Application developers can normally determine any configuration variable by means of reading from the stream opened by a call to:

```
popen("command -p getconf variable", "r");
```

The *confstr*( ) function with a *name* argument of _CS_PATH returns a string that can be used as a *PATH* environment variable setting that will reference the standard shell and utilities as described in the Shell and Utilities volume of IEEE Std 1003.1-200x.

The *confstr*( ) function copies the returned string into a buffer supplied by the application instead of returning a pointer to a string. This allows a cleaner function in some implementations (such as those with lightweight threads) and resolves questions about when the application must copy the string returned.

FUTURE DIRECTIONS

None.

SEE ALSO

pathconf( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *c99*

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

A table indicating the permissible values of *name* are added to the DESCRIPTION. All those marked EX are new in this issue.

Issue 6

The Open Group Corrigendum U033/7 is applied. The return value for the case returning the size of the buffer now explicitly states that this includes the terminating null.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated with new arguments which can be used to determine configuration strings for C compiler flags, linker/loader flags, and libraries for each different supported programming environment. This is a change to support data size neutrality.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is updated to include text describing how _CS_PATH can be used to obtain a *PATH* to access the standard utilities.

The macros associated with the *c89* programming models are marked LEGACY and new equivalent macros associated with *c99* are introduced.

NAME

conj, conjf, conjl — complex conjugate functions

SYNOPSIS

```
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex conjugate of *z*, by reversing the sign of its imaginary part.

RETURN VALUE

These functions return the complex conjugate value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *cproj*( ), *creal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

connect — connect a socket

SYNOPSIS

```
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address,
    socklen_t address_len);
```

DESCRIPTION

The *connect*( ) function shall attempt to make a connection on a socket. The function takes the following arguments:

socket	Specifies the file descriptor associated with the socket.
address	Points to a **sockaddr** structure containing the peer address. The length and format of the address depend on the address family of the socket.
address_len	Specifies the length of the **sockaddr** structure pointed to by the *address* argument.

If the socket has not already been bound to a local address, *connect*( ) shall bind it to an address which, unless the socket's address family is AF_UNIX, is an unused local address.

If the initiating socket is not connection-mode, then *connect*( ) shall set the socket's peer address, and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent *send*( ) functions, and limits the remote sender for subsequent *recv*( ) functions. If *address* is a null address for the protocol, the socket's peer address shall be reset.

If the initiating socket is connection-mode, then *connect*( ) shall attempt to establish a connection to the address specified by the *address* argument. If the connection cannot be established immediately and O_NONBLOCK is not set for the file descriptor for the socket, *connect*( ) shall block for up to an unspecified timeout interval until the connection is established. If the timeout interval expires before the connection is established, *connect*( ) shall fail and the connection attempt shall be aborted. If *connect*( ) is interrupted by a signal that is caught while blocked waiting to establish a connection, *connect*( ) shall fail and set *errno* to [EINTR], but the connection request shall not be aborted, and the connection shall be established asynchronously.

If the connection cannot be established immediately and O_NONBLOCK is set for the file descriptor for the socket, *connect*( ) shall fail and set *errno* to [EINPROGRESS], but the connection request shall not be aborted, and the connection shall be established asynchronously. Subsequent calls to *connect*( ) for the same socket, before the connection is established, shall fail and set *errno* to [EALREADY].

When the connection has been established asynchronously, *select*( ) and *poll*( ) shall indicate that the file descriptor for the socket is ready for writing.

The socket in use may require the process to have appropriate privileges to use the *connect*( ) function.

RETURN VALUE

Upon successful completion, *connect*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *connect*( ) function shall fail if:

[EADDRNOTAVAIL]
The specified address is not available from the local machine.

[EAFNOSUPPORT]	The specified address is not a valid address for the address family of the specified socket.
[EALREADY]	A connection request is already in progress for the specified socket.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNREFUSED]	The target address was not listening for connections or refused the connection request.
[EINPROGRESS]	O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.
[EINTR]	The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection shall be established asynchronously.
[EISCONN]	The specified socket is connection-mode and is already connected.
[ENETUNREACH]	No route to the network is present.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EPROTOTYPE]	The specified address has a different type than the socket bound to the specified peer address.
[ETIMEDOUT]	The attempt to connect timed out before a connection was made.

If the address family of the socket is AF_UNIX, then *connect*( ) shall fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in *address*.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
[ENOTDIR]	A component of the path prefix of the pathname in *address* is not a directory.

The *connect*( ) function may fail if:

[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
[EADDRINUSE]	Attempt to establish a connection that uses addresses that are already in use.
[ECONNRESET]	Remote host reset the connection request.
[EHOSTUNREACH]	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
[EINVAL]	The *address_len* argument is not a valid length for the address family; or invalid address family in the **sockaddr** structure.

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in *address*.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
[ENETDOWN]	The local network interface used to reach the destination is down.
[ENOBUFS]	No buffer space is available.
[EOPNOTSUPP]	The socket is listening and cannot be connected.

EXAMPLES

None.

APPLICATION USAGE

If *connect*( ) fails, the state of the socket is unspecified. Conforming applications should close the file descriptor and create a new socket before attempting to reconnect.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *close*( ), *getsockname*( ), *poll*( ), *select*( ), *send*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

copysign, copysignf, copysignl — number manipulation function

SYNOPSIS

```
#include <math.h>

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall produce a value with the magnitude of *x* and the sign of *y*. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, these functions regard the sign of zero as positive.

RETURN VALUE

Upon successful completion, these functions shall return a value with the magnitude of *x* and the sign of *y*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

signbit( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

cos, cosf, cosl — cosine function

SYNOPSIS

```
#include <math.h>

double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the cosine of their argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the cosine of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, the value 1.0 shall be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX

Domain Error	The *x* argument is ±Inf.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

Taking the Cosine of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45 * M_PI / 180;
double result;
...
result = cos(radians);
```

APPLICATION USAGE

These functions may lose accuracy when their argument is near an odd multiple of π/2 or is far from 0.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acos( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *sin*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *cosf*( ) and *cosl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

cosf — cosine function

SYNOPSIS

```
#include <math.h>

float cosf(float x);
```

DESCRIPTION

Refer to *cos*( ).

NAME

cosh, coshf, coshl — hyperbolic cosine functions

SYNOPSIS

```
#include <math.h>

double cosh(double x);
float coshf(float x);
long double coshl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the hyperbolic cosine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic cosine of *x*.

If the correct value would cause overflow, a range error shall occur and *cosh*( ), *coshf*( ), and *coshl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, the value 1.0 shall be returned.

If *x* is ±Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

Range Error	The result would cause an overflow. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

For IEEE Std 754-1985 **double**, $710.5 < |x|$ implies that *cosh*(*x*) has overflowed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

acosh( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *sinh*( ), *tanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *coshf*( ) and *coshl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

cosl — cosine function

SYNOPSIS

```
#include <math.h>

long double cosl(long double x);
```

DESCRIPTION

Refer to *cos*( ).

NAME

cpow, cpowf, cpowl — complex power functions

SYNOPSIS

```
#include <complex.h>

double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
    long double complex y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

RETURN VALUE

These functions shall return the complex power function value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs( ), *csqrt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

cproj, cprojf, cprojl — complex projection functions

SYNOPSIS

```
#include <complex.h>

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, then *cproj*(*z*) shall be equivalent to: |

```
INFINITY + I * copysign(0.0, cimag(z))
```

RETURN VALUE

These functions shall return the value of the projection onto the Riemann sphere.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Two topologies are commonly used in complex mathematics: the complex plane with its continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. The complex types with their multiplicity of infinities provide a useful (though imperfect) model for the complex plane. The *cproj*( ) function helps model the Riemann sphere by mapping all infinities to one, and should be used just before any operation, especially comparisons, that might give spurious results for any of the other infinities. Note that a complex value with one infinite part and one NaN part is regarded as an infinity, not a NaN, because if one part is infinite, the complex value is infinite independent of the value of the other part. For the same reason, *cabs*( ) returns an infinity if its argument has an infinite part and a NaN part.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *conj*( ), *creal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

creal, crealf, creall — complex real functions

SYNOPSIS

```
#include <complex.h>

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the real part of *z*.

RETURN VALUE

These functions shall return the real part value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For a variable *z* of complex type:

```
z == creal(z) + cimag(z)*I
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

carg( ), *cimag*( ), *conj*( ), *cproj*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

creat — create a new file or rewrite an existing one

SYNOPSIS

OH

```
#include <sys/stat.h>
#include <fcntl.h>

int creat(const char *path, mode_t mode);
```

DESCRIPTION

The function call:

```
creat(path, mode)
```

shall be equivalent to:

```
open(path, O_WRONLY|O_CREAT|O_TRUNC, mode)
```

RETURN VALUE

Refer to *open*( ).

ERRORS

Refer to *open*( ).

EXAMPLES

Creating a File

The following example creates the file **/tmp/file** with read and write permissions for the file owner and read permission for group and others. The resulting file descriptor is assigned to the *fd* variable.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = creat(filename, mode);
...
```

APPLICATION USAGE

None.

RATIONALE

The *creat*( ) function is redundant. Its services are also provided by the *open*( ) function. It has been included primarily for historical purposes since many existing applications depend on it. It is best considered a part of the C binding rather than a function that should be provided in other languages.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

crypt — string encoding function (**CRYPT**)

SYNOPSIS

XSI

```
#include <unistd.h>

char *crypt(const char *key, const char *salt);

```

DESCRIPTION

The *crypt*( ) function is a string encoding function. The algorithm is implementation-defined.

The *key* argument points to a string to be encoded. The *salt* argument is a string chosen from the set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . /
```

The first two characters of this string may be used to perturb the encoding algorithm.

The return value of *crypt*( ) points to static data that is overwritten by each call.

The *crypt*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *crypt*( ) shall return a pointer to the encoded string. The first two characters of the returned value shall be those of the *salt* argument. Otherwise, it shall return a null pointer and set *errno* to indicate the error. |

ERRORS

The *crypt*( ) function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

Encoding Passwords

The following example finds a user database entry matching a particular user name and changes the current password to a new password. The *crypt*( ) function generates an encoded version of each password. The first call to *crypt*( ) produces an encoded version of the old password; that encoded password is then compared to the password stored in the user database. The second call to *crypt*( ) encodes the new password before it is stored. |

The *putpwent*( ) function, used in the following example, is not part of IEEE Std 1003.1-200x.

```
#include <unistd.h>
#include <pwd.h>
#include <string.h>
#include <stdio.h>
...
int valid_change;
int pfd;  /* Integer for file descriptor returned by open(). */
FILE *fpfd;  /* File pointer for use in putpwent(). */
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
```

```
        char savepasswd[100];
        ...
        valid_change = 0;
        while ((p = getpwent()) != NULL) {
            /* Change entry if found. */
            if (strcmp(p->pw_name, user) == 0) {
                if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
                    strcpy(savepasswd, crypt(newpasswd, user));
                    p->pw_passwd = savepasswd;
                    valid_change = 1;
                }
                else {
                    fprintf(stderr, "Old password is not valid\n");
                }
            }
            /* Put passwd entry into ptmp. */
            putpwent(p, fpfd);
        }
```

APPLICATION USAGE

The values returned by this function need not be portable among XSI-conformant systems.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

encrypt( ), *setkey*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

NAME

csin, csinf, csinl — complex sine functions

SYNOPSIS

```
#include <complex.h>

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex sine of *z*.

RETURN VALUE

These functions shall return the complex sine value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

casin( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

csinf — complex sine functions

SYNOPSIS

```
#include <complex.h>

float complex csinf(float complex z);
```

DESCRIPTION

Refer to *csin*( ).

NAME

csinh, csinhf, csinhl — complex hyperbolic sine functions

SYNOPSIS

```
#include <complex.h>

double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex hyperbolic sine of *z*.

RETURN VALUE

These functions shall return the complex hyperbolic sine value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

casinh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

csinl — complex sine functions

SYNOPSIS

```
#include <complex.h>

long double complex csinl(long double complex z);
```

DESCRIPTION

Refer to *csin*( ).

NAME

csqrt, csqrtf, csqrtl — complex square root functions

SYNOPSIS

```
#include <complex.h>

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex square root of *z*, with a branch cut along the negative real axis.

RETURN VALUE

These functions shall return the complex square root value, in the range of the right half-plane (including the imaginary axis).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cabs( ), *cpow*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

ctan, ctanf, ctanl — complex tangent functions

SYNOPSIS

```
#include <complex.h>

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex tangent of *z*.

RETURN VALUE

These functions shall return the complex tangent value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catan( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

ctanf — complex tangent functions

SYNOPSIS

```
#include <complex.h>

float complex ctanf(float complex z);
```

DESCRIPTION

Refer to *ctan*( ).

NAME

ctanh, ctanhf, ctanhl — complex hyperbolic tangent functions

SYNOPSIS

```
#include <complex.h>

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complex hyperbolic tangent of *z*.

RETURN VALUE

These functions shall return the complex hyperbolic tangent value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

catanh( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<complex.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

ctanl — complex tangent functions

SYNOPSIS

```
#include <complex.h>

long double complex ctanl(long double complex z);
```

DESCRIPTION

Refer to *ctan*( ).

NAME

ctermid — generate a pathname for controlling terminal

SYNOPSIS

CX
```
#include <stdio.h>

char *ctermid(char *s);
```

DESCRIPTION

The *ctermid*( ) function shall generate a string that, when used as a pathname, refers to the current controlling terminal for the current process. If *ctermid*( ) returns a pathname, access to the file is not guaranteed.

If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, it shall ensure that the *ctermid*( ) function is called with a non-NULL parameter.

RETURN VALUE

If *s* is a null pointer, the string shall be generated in an area that may be static (and therefore may be overwritten by each call), the address of which shall be returned. Otherwise, *s* is assumed to point to a character array of at least L_ctermid bytes; the string is placed in this array and the value of *s* shall be returned. The symbolic constant L_ctermid is defined in **<stdio.h>**, and shall have a value greater than 0.

The *ctermid*( ) function shall return an empty string if the pathname that would refer to the controlling terminal cannot be determined, or if the function is unsuccessful.

ERRORS

No errors are defined.

EXAMPLES

Determining the Controlling Terminal for the Current Process

The following example returns a pointer to a string that identifies the controlling terminal for the current process. The pathname for the terminal is stored in the array pointed to by the *ptr* argument, which has a size of L_ctermid bytes, as indicated by the *term* argument.

```
#include <stdio.h>
...
char term[L_ctermid];
char *ptr;

ptr = ctermid(term);
```

APPLICATION USAGE

The difference between *ctermid*( ) and *ttyname*( ) is that *ttyname*( ) must be handed a file descriptor and return a path of the terminal associated with that file descriptor, while *ctermid*( ) returns a string (such as `"/dev/tty"`) that refers to the current controlling terminal if used as a pathname.

RATIONALE

L_ctermid must be defined appropriately for a given implementation and must be greater than zero so that array declarations using it are accepted by the compiler. The value includes the terminating null byte.

Conforming applications that use threads cannot call *ctermid*( ) with NULL as the parameter if either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined. If *s* is not NULL, the *ctermid*( ) function generates a string that, when used as a pathname, refers to the current controlling terminal for the current process. If *s* is NULL, the return value of *ctermid*( ) is undefined.

There is no additional burden on the programmer—changing to use a hypothetical thread-safe version of *ctermid*( ) along with allocating a buffer is more of a burden than merely allocating a buffer. Application code should not assume that the returned string is short, as some implementations have more than two pathname components before reaching a logical device name.

FUTURE DIRECTIONS

None.

SEE ALSO

ttyname( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

ctime, ctime_r — convert a time value to date and time string

SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);
char *ctime_r(const time_t *clock, char *buf);
```

TSF (ctime_r)

DESCRIPTION

CX For *ctime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ctime*( ) function shall convert the time pointed to by *clock*, representing time in seconds since the Epoch, to local time in the form of a string. It shall be equivalent to:

```
asctime(localtime(clock))
```

CX The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static objects: a broken-down time structure and an array of **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The *ctime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *ctime_r*( ) function shall convert the calendar time pointed to by *clock* to local time in exactly the same form as *ctime*( ) and puts the string into the array pointed to by *buf* (which shall be at least 26 bytes in size) and return *buf*.

Unlike *ctime*( ), the thread-safe version *ctime_r*( ) is not required to set *tzname*.

RETURN VALUE

The *ctime*( ) function shall return the pointer returned by *asctime*( ) with that broken-down time as an argument.

TSF Upon successful completion, *ctime_r*( ) shall return a pointer to the string pointed to by *buf*. When an error is encountered, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Values for the broken-down time structure can be obtained by calling *gmtime*( ) or *localtime*( ). The *ctime*( ) function is included for compatibility with older implementations, and does not support localized date and time formats. Applications should use the *strftime*( ) function to achieve maximum portability.

The *ctime_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

The *ctime_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *ctime*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

NAME

daylight — daylight savings time flag

SYNOPSIS

XSI

```
#include <time.h>

extern int daylight;
```

DESCRIPTION

Refer to *tzset*( ).

NAME

dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, dbm_nextkey, dbm_open, dbm_store — database functions

SYNOPSIS

XSI
```
#include <ndbm.h>

int dbm_clearerr(DBM *db);
void dbm_close(DBM *db);
int dbm_delete(DBM *db, datum key);
int dbm_error(DBM *db);
datum dbm_fetch(DBM *db, datum key);
datum dbm_firstkey(DBM *db);
datum dbm_nextkey(DBM *db);
DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

DESCRIPTION

These functions create, access, and modify a database.

A **datum** consists of at least two members, *dptr* and *dsize*. The *dptr* member points to an object that is *dsize* bytes in length. Arbitrary binary data, as well as character strings, may be stored in the object pointed to by *dptr*.

The database is stored in two files. One file is a directory containing a bit map of keys and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

The *dbm_open*( ) function shall open a database. The *file* argument to the function is the pathname of the database. The function opens two files named *file*.**dir** and *file*.**pag**. The *open_flags* argument has the same meaning as the *flags* argument of *open*( ) except that a database opened for write-only access opens the files for read and write access and the behavior of the O_APPEND flag is unspecified. The *file_mode* argument has the same meaning as the third argument of *open*( ).

The *dbm_close*( ) function shall close a database. The application shall ensure that argument *db* is a pointer to a **dbm** structure that has been returned from a call to *dbm_open*( ).

These database functions shall support an internal block size large enough to support key/content pairs of at least 1 023 bytes.

The *dbm_fetch*( ) function shall read a record from a database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that matches the key of the record the program is fetching.

The *dbm_store*( ) function shall write a record to a database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that identifies (for subsequent reading, writing, or deleting) the record the application is writing. The argument *content* is a **datum** that has been initialized by the application to the value of the record the program is writing. The argument *store_mode* controls whether *dbm_store*( ) replaces any pre-existing record that has the same key that is specified by the *key* argument. The application shall set *store_mode* to either DBM_INSERT or DBM_REPLACE. If the database contains a record that matches the *key* argument and *store_mode* is DBM_REPLACE, the existing record shall be replaced with the new record. If the database contains a record that matches the *key* argument and *store_mode* is DBM_INSERT, the existing record shall be left unchanged and the new record ignored. If the database does not contain a record that matches the *key* argument and *store_mode* is either DBM_INSERT or DBM_REPLACE, the new record shall be inserted in the database.

If the sum of a key/content pair exceeds the internal block size, the result is unspecified. Moreover, the application shall ensure that all key/content pairs that hash together fit on a single block. The *dbm_store*( ) function shall return an error in the event that a disk block fills with inseparable data.

The *dbm_delete*( ) function shall delete a record and its key from the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The argument *key* is a **datum** that has been initialized by the application to the value of the key that identifies the record the program is deleting.

The *dbm_firstkey*( ) function shall return the first key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dbm_nextkey*( ) function shall return the next key in the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ). The application shall ensure that the *dbm_firstkey*( ) function is called before calling *dbm_nextkey*( ). Subsequent calls to *dbm_nextkey*( ) return the next key until all of the keys in the database have been returned.

The *dbm_error*( ) function shall return the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dbm_clearerr*( ) function shall clear the error condition of the database. The argument *db* is a pointer to a database structure that has been returned from a call to *dbm_open*( ).

The *dptr* pointers returned by these functions may point into static storage that may be changed by subsequent calls.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *dbm_store*( ) and *dbm_delete*( ) functions shall return 0 when they succeed and a negative value when they fail.

The *dbm_store*( ) function shall return 1 if it is called with a *flags* value of DBM_INSERT and the function finds an existing record with the same key.

The *dbm_error*( ) function shall return 0 if the error condition is not set and return a non-zero value if the error condition is set.

The return value of *dbm_clearerr*( ) is unspecified.

The *dbm_firstkey*( ) and *dbm_nextkey*( ) functions shall return a key **datum**. When the end of the database is reached, the *dptr* member of the key is a null pointer. If an error is detected, the *dptr* member of the key shall be a null pointer and the error condition of the database shall be set.

The *dbm_fetch*( ) function shall return a content **datum**. If no record in the database matches the key or if an error condition has been detected in the database, the *dptr* member of the content shall be a null pointer.

The *dbm_open*( ) function shall return a pointer to a database structure. If an error is detected during the operation, *dbm_open*( ) shall return a (**DBM ***)0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following code can be used to traverse the database:

```
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

The *dbm_* functions provided in this library should not be confused in any way with those of a general-purpose database management system. These functions do not provide for multiple search keys per entry, they do not protect against multi-user access (in other words they do not lock records or files), and they do not provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions are useful for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The *dbm_delete*( ) function need not physically reclaim file space, although it does make it available for reuse by the database.

After calling *dbm_store*( ) or *dbm_delete*( ) during a pass through the keys by *dbm_firstkey*( ) and *dbm_nextkey*( ), the application should reset the database by calling *dbm_firstkey*( ) before again calling *dbm_nextkey*( ). The contents of these files are unspecified and may not be portable.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ndbm.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

difftime — compute the difference between two calendar time values

SYNOPSIS

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *difftime*( ) function shall compute the difference between two calendar times (as returned by *time*( )): *time1* − *time0*.

RETURN VALUE

The *difftime*( ) function shall return the difference expressed in seconds as a type **double**.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

NAME

dirname — report the parent directory name of a file pathname |

SYNOPSIS

XSI
```
#include <libgen.h>

char *dirname(char *path);
```

DESCRIPTION

The *dirname*( ) function shall take a pointer to a character string that contains a pathname, and return a pointer to a string that is a pathname of the parent directory of that file. Trailing `'/'` characters in the path are not counted as part of the path. |

If *path* does not contain a `'/'`, then *dirname*( ) shall return a pointer to the string `"."`. If *path* is a null pointer or points to an empty string, *dirname*( ) shall return a pointer to the string `"."`.

The *dirname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *dirname*( ) function shall return a pointer to a string that is the parent directory of *path*. If *path* is a null pointer or points to an empty string, a pointer to a string `"."` is returned.

The *dirname*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten by subsequent calls to *dirname*( ).

ERRORS

No errors are defined.

EXAMPLES

The following code fragment reads a pathname, changes the current working directory to the parent directory, and opens the file. |

```
char path[MAXPATHLEN], *pathcopy;
int fd;
fgets(path, MAXPATHLEN, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

Sample Input and Output Strings for dirname( )

In the following table, the input string is the value pointed to by *path*, and the output string is the return value of the *dirname*( ) function.

Input String	Output String
`"/usr/lib"`	`"/usr"`
`"/usr/"`	`"/"`
`"usr"`	`"."`
`"/"`	`"/"`
`"."`	`"."`
`".."`	`"."`

Changing the Current Directory to the Parent Directory

The following program fragment reads a pathname, changes the current working directory to the parent directory, and opens the file. |

```
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <libgen.h>
...
char path[PATH_MAX], *pathcopy;
int fd;
...
fgets(path, PATH_MAX, stdin);
pathcopy = strdup(path);
chdir(dirname(pathcopy));
fd = open(basename(path), O_RDONLY);
```

APPLICATION USAGE

The *dirname*( ) and *basename*( ) functions together yield a complete pathname. The expression *dirname*(*path*) obtains the pathname of the directory where *basename*(*path*) is found. |

Since the meaning of the leading `"//"` is implementation-defined, *dirname*("**//foo**) may return either `"//"` or `'/'` (but nothing else).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

basename( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<libgen.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

div — compute the quotient and remainder of an integer division

SYNOPSIS

```
#include <stdlib.h>

div_t div(int numer, int denom);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *div*( ) function shall compute the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, *quot***denom*+*rem* shall equal *numer*.

RETURN VALUE

The *div*( ) function shall return a structure of type **div_t**, comprising both the quotient and the remainder. The structure includes the following members, in any order:

```
int  quot;  /* quotient */
int  rem;   /* remainder */
```

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ldiv( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

NAME

dlclose — close a *dlopen*( ) object

SYNOPSIS

XSI

```
#include <dlfcn.h>

int dlclose(void *handle);
```

DESCRIPTION

The *dlclose*( ) function shall inform the system that the object referenced by a *handle* returned from a previous *dlopen*( ) invocation is no longer needed by the application.

The use of *dlclose*( ) reflects a statement of intent on the part of the process, but does not create any requirement upon the implementation, such as removal of the code or symbols referenced by *handle*. Once an object has been closed using *dlclose*( ) an application should assume that its symbols are no longer available to *dlsym*( ). All objects loaded automatically as a result of invoking *dlopen*( ) on the referenced object shall also be closed if this is the last reference to it.

Although a *dlclose*( ) operation is not required to remove structures from an address space, neither is an implementation prohibited from doing so. The only restriction on such a removal is that no object shall be removed to which references have been relocated, until or unless all such references are removed. For instance, an object that had been loaded with a *dlopen*( ) operation specifying the RTLD_GLOBAL flag might provide a target for dynamic relocations performed in the processing of other objects—in such environments, an application may assume that no relocation, once made, shall be undone or remade unless the object requiring the relocation has itself been removed.

RETURN VALUE

If the referenced object was successfully closed, *dlclose*( ) shall return 0. If the object could not be closed, or if *handle* does not refer to an open object, *dlclose*( ) shall return a non-zero value. More detailed diagnostic information shall be available through *dlerror*( ).

ERRORS

No errors are defined.

EXAMPLES

The following example illustrates use of *dlopen*( ) and *dlclose*( ):

```
...
/* Open a dynamic library and then close it ... */

#include <dlfcn.h>
void *mylib;
int eret;

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

APPLICATION USAGE

A conforming application should employ a *handle* returned from a *dlopen*( ) invocation only within a given scope bracketed by the *dlopen*( ) and *dlclose*( ) operations. Implementations are free to use reference counting or other techniques such that multiple calls to *dlopen*( ) referencing the same object may return the same object for *handle*. Implementations are also free to reuse a *handle*. For these reasons, the value of a *handle* must be treated as an opaque object by the application, used only in calls to *dlsym*( ) and *dlclose*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlerror( ), *dlopen*( ), *dlsym*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The DESCRIPTION is updated to say that the referenced object is closed ‘‘if this is the last reference to it’’.

NAME

dlerror — get diagnostic information

SYNOPSIS

XSI

```
#include <dlfcn.h>

char *dlerror(void);
```

DESCRIPTION

The *dlerror*( ) function shall return a null-terminated character string (with no trailing <newline>) that describes the last error that occurred during dynamic linking processing. If no dynamic linking errors have occurred since the last invocation of *dlerror*( ), *dlerror*( ) shall return NULL. Thus, invoking *dlerror*( ) a second time, immediately following a prior invocation, shall result in NULL being returned.

The *dlerror*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If successful, *dlerror*( ) shall return a null-terminated character string; otherwise, NULL shall be returned.

ERRORS

No errors are defined.

EXAMPLES

The following example prints out the last dynamic linking error:

```
...
#include <dlfcn.h>

char *errstr;

errstr = dlerror();
if (errstr != NULL)
printf ("A dynamic linking error occurred: (%s)\n", errstr);
...
```

APPLICATION USAGE

The messages returned by *dlerror*( ) may reside in a static buffer that is overwritten on each call to *dlerror*( ). Application code should not write to this buffer. Programs wishing to preserve an error message should make their own copies of that message. Depending on the application environment with respect to asynchronous execution events, such as signals or other asynchronous computation sharing the address space, conforming applications should use a critical section to retrieve the error pointer and buffer. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlclose( ), *dlopen*( ), *dlsym*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

In the DESCRIPTION the note about reentrancy and thread-safety is added.

NAME

dlopen — gain access to an executable object file

SYNOPSIS

XSI

```
#include <dlfcn.h>

void *dlopen(const char *file, int mode);
```

DESCRIPTION

The *dlopen*( ) function shall make an executable object file specified by *file* available to the calling program. The class of files eligible for this operation and the manner of their construction are implementation-defined, though typically such files are executable objects such as shared libraries, relocatable files, or programs. Note that some implementations permit the construction of dependencies between such objects that are embedded within files. In such cases, a *dlopen*( ) operation shall load such dependencies in addition to the object referenced by *file*. Implementations may also impose specific constraints on the construction of programs that can employ *dlopen*( ) and its related services.

A successful *dlopen*( ) shall return a *handle* which the caller may use on subsequent calls to *dlsym*( ) and *dlclose*( ). The value of this *handle* should not be interpreted in any way by the caller.

The *file* argument is used to construct a pathname to the object file. If *file* contains a slash character, the *file* argument is used as the pathname for the file. Otherwise, *file* is used in an implementation-defined manner to yield a pathname.

If the value of *file* is 0, *dlopen*( ) shall provide a *handle* on a global symbol object. This object shall provide access to the symbols from an ordered set of objects consisting of the original program image file, together with any objects loaded at program start-up as specified by that process image file (for example, shared libraries), and the set of objects loaded using a *dlopen*( ) operation together with the RTLD_GLOBAL flag. As the latter set of objects can change during execution, the set identified by *handle* can also change dynamically.

Only a single copy of an object file is brought into the address space, even if *dlopen*( ) is invoked multiple times in reference to the file, and even if different pathnames are used to reference the file.

The *mode* parameter describes how *dlopen*( ) shall operate upon *file* with respect to the processing of relocations and the scope of visibility of the symbols provided within *file*. When an object is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references shall be relocated before the symbols can be accessed. The *mode* parameter governs when these relocations take place and may have the following values:

RTLD_LAZY	Relocations shall be performed at an implementation-defined time, ranging from the time of the *dlopen*( ) call until the first reference to a given symbol occurs. Specifying RTLD_LAZY should improve performance on implementations supporting dynamic symbol binding as a process may not reference all of the functions in any given object. And, for systems supporting dynamic symbol resolution for normal process execution, this behavior mimics the normal handling of process execution.
RTLD_NOW	All necessary relocations shall be performed when the object is first loaded. This may waste some processing if relocations are performed for functions that are never referenced. This behavior may be useful for applications that need to know as soon as an object is loaded that all

symbols referenced during execution are available.

Any object loaded by *dlopen*( ) that requires relocations against global symbols can reference the symbols in the original process image file, any objects loaded at program start-up, from the object itself as well as any other object included in the same *dlopen*( ) invocation, and any objects that were loaded in any *dlopen*( ) invocation and which specified the RTLD_GLOBAL flag. To determine the scope of visibility for the symbols loaded with a *dlopen*( ) invocation, the *mode* parameter should be a bitwise-inclusive OR with one of the following values:

RTLD_GLOBAL	The object's symbols shall be made available for the relocation processing of any other object. In addition, symbol lookup using *dlopen*(*0*, *mode*) and an associated *dlsym*( ) allows objects loaded with this *mode* to be searched.
RTLD_LOCAL	The object's symbols shall not be made available for the relocation processing of any other object.

If neither RTLD_GLOBAL nor RTLD_LOCAL are specified, then an implementation-defined default behavior shall be applied.

If a *file* is specified in multiple *dlopen*( ) invocations, *mode* is interpreted at each invocation. Note, however, that once RTLD_NOW has been specified all relocations shall have been completed rendering further RTLD_NOW operations redundant and any further RTLD_LAZY operations irrelevant. Similarly, note that once RTLD_GLOBAL has been specified the object shall maintain the RTLD_GLOBAL status regardless of any previous or future specification of RTLD_LOCAL, as long as the object remains in the address space (see *dlclose*( )).

Symbols introduced into a program through calls to *dlopen*( ) may be used in relocation activities. Symbols so introduced may duplicate symbols already defined by the program or previous *dlopen*( ) operations. To resolve the ambiguities such a situation might present, the resolution of a symbol reference to symbol definition is based on a symbol resolution order. Two such resolution orders are defined: *load* or *dependency* ordering. Load order establishes an ordering among symbol definitions, such that the definition first loaded (including definitions from the image file and any dependent objects loaded with it) has priority over objects added later (via *dlopen*( )). Load ordering is used in relocation processing. Dependency ordering uses a breadth-first order starting with a given object, then all of its dependencies, then any dependents of those, iterating until all dependencies are satisfied. With the exception of the global symbol object obtained via a *dlopen*( ) operation on a *file* of 0, dependency ordering is used by the *dlsym*( ) function. Load ordering is used in *dlsym*( ) operations upon the global symbol object.

When an object is first made accessible via *dlopen*( ) it and its dependent objects are added in dependency order. Once all the objects are added, relocations are performed using load order. Note that if an object or its dependencies had been previously loaded, the load and dependency orders may yield different resolutions.

The symbols introduced by *dlopen*( ) operations, and available through *dlsym*( ) are at a minimum those which are exported as symbols of global scope by the object. Typically such symbols shall be those that were specified in (for example) C source code as having *extern* linkage. The precise manner in which an implementation constructs the set of exported symbols for a *dlopen*( ) object is specified by that implementation.

RETURN VALUE

If *file* cannot be found, cannot be opened for reading, is not of an appropriate object format for processing by *dlopen*( ), or if an error occurs during the process of loading *file* or relocating its symbolic references, *dlopen*( ) shall return NULL. More detailed diagnostic information shall be available through *dlerror*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlclose( ), *dlerror*( ), *dlsym*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

NAME

dlsym — obtain the address of a symbol from a *dlopen*( ) object

SYNOPSIS

XSI

```
#include <dlfcn.h>

void *dlsym(void *restrict handle, const char *restrict name);
```

DESCRIPTION

The *dlsym*( ) function shall obtain the address of a symbol defined within an object made accessible through a *dlopen*( ) call. The *handle* argument is the value returned from a call to *dlopen*( ) (and which has not since been released via a call to *dlclose*( )), and *name* is the symbol's name as a character string.

The *dlsym*( ) function shall search for the named symbol in all objects loaded automatically as a result of loading the object referenced by *handle* (see *dlopen*( )). Load ordering is used in *dlsym*( ) operations upon the global symbol object. The symbol resolution algorithm used shall be dependency order as described in *dlopen*( ).

The RTLD_NEXT flag is reserved for future use.

RETURN VALUE

If *handle* does not refer to a valid object opened by *dlopen*( ), or if the named symbol cannot be found within any of the objects associated with *handle*, *dlsym*( ) shall return NULL. More detailed diagnostic information shall be available through *dlerror*( ).

ERRORS

No errors are defined.

EXAMPLES

The following example shows how *dlopen*( ) and *dlsym*( ) can be used to access either function or data objects. For simplicity, error checking has been omitted.

```
void    *handle;
int     *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

APPLICATION USAGE

Special purpose values for *handle* are reserved for future use. These values and their meanings are:

RTLD_DEFAULT	The symbol lookup happens in the normal global scope; that is, a search for a symbol using this handle would find the same definition as a direct use of this symbol in the program code.
RTLD_NEXT	Specifies the next object after this one that defines *name*. *This one* refers to the object containing the invocation of *dlsym*( ). The *next* object is the one found upon the application of a load order symbol resolution algorithm (see *dlopen*( )). The next object is either one of global scope (because it was introduced as part of the original process image or because it was added with

a *dlopen*( ) operation including the RTLD_GLOBAL flag), or is an object that was included in the same *dlopen*( ) operation that loaded this one.

The RTLD_NEXT flag is useful to navigate an intentionally created hierarchy of multiply-defined symbols created through *interposition*. For example, if a program wished to create an implementation of *malloc*( ) that embedded some statistics gathering about memory allocations, such an implementation could use the real *malloc*( ) definition to perform the memory allocation—and itself only embed the necessary logic to implement the statistics gathering function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

dlclose( ), *dlerror*( ), *dlopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dlfcn.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The **restrict** keyword is added to the *dlsym*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48 — generate uniformly distributed pseudo-random numbers

SYNOPSIS

XSI
```
#include <stdlib.h>

double drand48(void);
double erand48(unsigned short xsubi[3]);
long jrand48(unsigned short xsubi[3]);
void lcong48(unsigned short param[7]);
long lrand48(void);
long mrand48(void);
long nrand48(unsigned short xsubi[3]);
unsigned short *seed48(unsigned short seed16v[3]);
void srand48(long seedval);
```

DESCRIPTION

This family of functions shall generate pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The *drand48*( ) and *erand48*( ) functions shall return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The *lrand48*( ) and *nrand48*( ) functions shall return non-negative, long integers, uniformly distributed over the interval $[0,2^{31})$.

The *mrand48*( ) and *jrand48*( ) functions shall return signed long integers uniformly distributed over the interval $[-2^{31},2^{31})$.

The *srand48*( ), *seed48*( ), and *lcong48*( ) are initialization entry points, one of which should be invoked before either *drand48*( ), *lrand48*( ), or *mrand48*( ) is called. (Although it is not recommended practice, constant default initializer values shall be supplied automatically if *drand48*( ), *lrand48*( ), or *mrand48*( ) is called without a prior call to an initialization entry point.) The *erand48*( ), *nrand48*( ), and *jrand48*( ) functions do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\ m} \qquad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *lcong48*( ) is invoked, the multiplier value a and the addend value c are given by:

$$a = \mathrm{5DEECE66D}_{16} = 273673163155_{8}$$

$$c = \mathrm{B}_{16} = 13_{8}$$

The value returned by any of the *drand48*( ), *erand48*( ), *jrand48*( ), *lrand48*( ), *mrand48*( ), or *nrand48*( ) functions is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The *drand48*( ), *lrand48*( ), and *mrand48*( ) functions store the last 48-bit X_i generated in an internal buffer; that is why the application shall ensure that these are initialized prior to being invoked. The *erand48*( ), *nrand48*( ), and *jrand48*( ) functions require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, *erand48*( ), *nrand48*( ), and *jrand48*( ) allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers; that is, the sequence of numbers in each stream shall *not* depend upon how many times the routines are called to generate numbers for the other streams.

The initializer function *srand48*( ) sets the high-order 32 bits of X_i to the low-order 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function *seed48*( ) sets the value of X_i to the 48-bit value specified in the argument array. The low-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*0*]. The mid-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*1*]. The high-order 16 bits of X_i are set to the low-order 16 bits of *seed16v*[*2*]. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by *seed48*( ), and a pointer to this buffer is the value returned by *seed48*( ). This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time—use the pointer to get at and store the last X_i value, and then use this value to reinitialize via *seed48*( ) when the program is restarted.

The initializer function *lcong48*( ) allows the user to specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array elements *param*[*0-2*] specify X_i, *param*[*3-5*] specify the multiplier *a*, and *param*[*6*] specifies the 16-bit addend *c*. After *lcong48*( ) is called, a subsequent call to either *srand48*( ) or *seed48*( ) shall restore the standard multiplier and addend values, *a* and *c*, specified above.

The *drand48*( ), *lrand48*( ), and *mrand48*( ) functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

As described in the DESCRIPTION above.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

rand( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

dup, dup2 — duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup(int fildes);
int dup2(int fildes, int fildes2);
```

DESCRIPTION

The *dup*( ) and *dup2*( ) functions provide an alternative interface to the service provided by *fcntl*( ) using the F_DUPFD command. The call:

```
fid = dup(fildes);
```

shall be equivalent to:

```
fid = fcntl(fildes, F_DUPFD, 0);
```

The call:

```
fid = dup2(fildes, fildes2);
```

shall be equivalent to:

```
close(fildes2);
fid = fcntl(fildes, F_DUPFD, fildes2);
```

except for the following:

- If *fildes2* is less than 0 or greater than or equal to {OPEN_MAX}, *dup2*( ) shall return −1 with *errno* set to [EBADF].
- If *fildes* is a valid file descriptor and is equal to *fildes2*, *dup2*( ) shall return *fildes2* without closing it.
- If *fildes* is not a valid file descriptor, *dup2*( ) shall return −1 and shall not close *fildes2*.
- The value returned shall be equal to the value of *fildes2* upon successful completion, or −1 upon failure.

RETURN VALUE

Upon successful completion a non-negative integer, namely the file descriptor, shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *dup*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EMFILE]	The number of file descriptors in use by this process would exceed {OPEN_MAX}.

The *dup2*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor or the argument *fildes2* is negative or greater than or equal to {OPEN_MAX}.
[EINTR]	The *dup2*( ) function was interrupted by a signal.

EXAMPLES

Redirecting Standard Output to a File

The following example closes standard output for the current processes, re-assigns standard output to go to the file referenced by *pfd*, and closes the original file descriptor to clean up.

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

Redirecting Error Messages

The following example redirects messages from *stderr* to *stdout*.

```
#include <unistd.h>
...
dup2(1, 2);
...
```

APPLICATION USAGE

None.

RATIONALE

The *dup*( ) and *dup2*( ) functions are redundant. Their services are also provided by the *fcntl*( ) function. They have been included in this volume of IEEE Std 1003.1-200x primarily for historical reasons, since many existing applications use them.

While the brief code segment shown is very similar in behavior to *dup2*( ), a conforming implementation based on other functions defined in this volume of IEEE Std 1003.1-200x is significantly more complex. Least obvious is the possible effect of a signal-catching function that could be invoked between steps and allocate or deallocate file descriptors. This could be avoided by blocking signals.

The *dup2*( ) function is not marked obsolescent because it presents a type-safe version of functionality provided in a type-unsafe version by *fcntl*( ). It is used in the POSIX Ada binding.

The *dup2*( ) function is not intended for use in critical regions as a synchronization mechanism.

In the description of [EBADF], the case of *fildes* being out of range is covered by the given case of *fildes* not being valid. The descriptions for *fildes* and *fildes2* are different because the only kind of invalidity that is relevant for *fildes2* is whether it is out of range; that is, it does not matter whether *fildes2* refers to an open file when the *dup2*( ) call is made.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *fcntl*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The RATIONALE section is added.

NAME

ecvt, fcvt, gcvt — convert a floating-point number to a string (**LEGACY**)

SYNOPSIS

XSI

```
#include <stdlib.h>

char *ecvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
char *fcvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
char *gcvt(double value, int ndigit, char *buf);
```

DESCRIPTION

The *ecvt*( ), *fcvt*( ), and *gcvt*( ) functions shall convert floating-point numbers to null-terminated strings.

The *ecvt*( ) function shall convert *value* to a null-terminated string of *ndigit* digits (where *ndigit* is reduced to an unspecified limit determined by the precision of a **double**) and return a pointer to the string. The high-order digit shall be non-zero, unless the value is 0. The low-order digit shall be rounded in an implementation-defined manner. The position of the radix character relative to the beginning of the string shall be stored in the integer pointed to by *decpt* (negative means to the left of the returned digits). If *value* is zero, it is unspecified whether the integer pointed to by *decpt* would be 0 or 1. The radix character shall not be included in the returned string. If the sign of the result is negative, the integer pointed to by *sign* shall be non-zero; otherwise, it shall be 0.

If the converted value is out of range or is not representable, the contents of the returned string are unspecified.

The *fcvt*( ) function shall be equivalent to *ecvt*( ), except that *ndigit* specifies the number of digits desired after the radix character. The total number of digits in the result string is restricted to an unspecified limit as determined by the precision of a **double**.

The *gcvt*( ) function shall convert *value* to a null-terminated string (similar to that of the `%g` conversion specification format of *printf*( )) in the array pointed to by *buf* and shall return *buf*. It shall produce *ndigit* significant digits (limited to an unspecified value determined by the precision of a **double**) in the `%f` conversion specification format of *printf*( ) if possible, or the `%e` conversion specification format of *printf*( ) (scientific notation) otherwise. A minus sign shall be included in the returned string if *value* is less than 0. A radix character shall be included in the returned string if *value* is not a whole number. Trailing zeros shall be suppressed where *value* is not a whole number. The radix character is determined by the current locale. If *setlocale*( ) has not been called successfully, the default locale, POSIX, is used. The default locale specifies a period (`'.'`) as the radix character. The *LC_NUMERIC* category determines the value of the radix character within the current locale.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *ecvt*( ) and *fcvt*( ) functions shall return a pointer to a null-terminated string of digits.

The *gcvt*( ) function shall return *buf*.

The return values from *ecvt*( ) and *fcvt*( ) may point to static data which may be overwritten by subsequent calls to these functions.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

sprintf( ) is preferred over this function.

RATIONALE

None.

FUTURE DIRECTIONS

These functions may be withdrawn in a future version.

SEE ALSO

printf( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

This function is marked LEGACY.

The **restrict** keyword is added to the *ecvt*( ) and *fcvt*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION is updated to explicitly use ‘‘conversion specification’’ to describe `%g`, `%f`, and `%e`.

NAME

encrypt — encoding function (**CRYPT**)

SYNOPSIS

XSI

```
#include <unistd.h>

void encrypt(char block[64], int edflag);
```

DESCRIPTION

The *encrypt*( ) function shall provide access to an implementation-defined encoding algorithm. The key generated by *setkey*( ) is used to encrypt the string *block* with *encrypt*( ).

The *block* argument to *encrypt*( ) shall be an array of length 64 bytes containing only the bytes with values of 0 and 1. The array is modified in place to a similar array using the key set by *setkey*( ). If *edflag* is 0, the argument is encoded. If *edflag* is 1, the argument may be decoded (see the APPLICATION USAGE section); if the argument is not decoded, *errno* shall be set to [ENOSYS].

The *encrypt*( ) function shall not change the setting of *errno* if successful. An application wishing to check for error situations should set *errno* to 0 before calling *encrypt*( ). If *errno* is non-zero on return, an error has occurred.

The *encrypt*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *encrypt*( ) function shall not return a value.

ERRORS

The *encrypt*( ) function shall fail if:

[ENOSYS] The functionality is not supported on this implementation.

EXAMPLES

None.

APPLICATION USAGE

Historical implementations of the *encrypt*( ) function used a rather primitive encoding algorithm.

In some environments, decoding might not be implemented. This is related to some Government restrictions on encryption and decryption routines. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of *encrypt*( ) does encoding but not decoding.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

crypt( ), *setkey*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

endgrent, getgrent, setgrent — group database entry functions

SYNOPSIS

XSI
```
#include <grp.h>

void endgrent(void);
struct group *getgrent(void);
void setgrent(void);
```

DESCRIPTION

The *getgrent*( ) function shall return a pointer to a structure containing the broken-out fields of an entry in the group database. When first called, *getgrent*( ) shall return a pointer to a **group** structure containing the first entry in the group database. Thereafter, it shall return a pointer to a **group** structure containing the next group structure in the group database, so successive calls may be used to search the entire database.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the group database. In particular, the system may deny the existence of some or all of the group database entries associated with groups other than those groups associated with the caller and may omit users other than the caller from the list of members of groups in database entries that are returned.

The *setgrent*( ) function shall rewind the group database to allow repeated searches. |

The *endgrent*( ) function may be called to close the group database when processing is complete.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

When first called, *getgrent*( ) shall return a pointer to the first group structure in the group database. Upon subsequent calls it shall return the next group structure in the group database. The *getgrent*( ) function shall return a null pointer on end-of-file or an error and *errno* may be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to *getgrgid*( ), *getgrnam*( ), or *getgrent*( ).

ERRORS

The *getgrent*( ) function may fail if:

[EINTR]	A signal was caught during the operation.
[EIO]	An I/O error has occurred.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

EXAMPLES

None.

APPLICATION USAGE

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the group database, whether the database is a single file, or where in the file system name space the database resides. Applications should use *getgrnam*( ) and *getgrgid*( ) whenever possible because it avoids these dependencies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getgrgid( ), *getgrnam*( ), *getlogin*( ), *getpwent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

endhostent, gethostent, sethostent — network host database functions

SYNOPSIS

```
#include <netdb.h>

void endhostent(void);
struct hostent *gethostent(void);
void sethostent(int stayopen);
```

DESCRIPTION

These functions shall retrieve information about hosts. This information is considered to be stored in a database that can be accessed sequentially or randomly. The implementation of this database is unspecified.

Note: In many cases it is implemented by the Domain Name System, as documented in RFC 1034, RFC 1035, and RFC 1886.

The *sethostent*( ) function shall open a connection to the database and set the next entry for retrieval to the first entry in the database. If the *stayopen* argument is non-zero, the connection shall not be closed by a call to *gethostent*( ), *gethostbyname*( ), or *gethostbyaddr*( ), and the implementation may maintain an open file descriptor.

The *gethostent*( ) function shall read the next entry in the database, opening and closing a connection to the database as necessary.

Entries shall be returned in **hostent** structures. Refer to *gethostbyaddr*( ) for a definition of the **hostent** structure.

The *endhostent*( ) function shall close the connection to the database, releasing any open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, the *gethostent*( ) function shall return a pointer to a **hostent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found.

ERRORS

No errors are defined for *endhostent*( ), *gethostent*( ), and *sethostent*( ).

EXAMPLES

None.

APPLICATION USAGE

The *gethostent*( ) function may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endservent( ), *gethostbyaddr*( ), *gethostbyname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endnetent, getnetbyaddr, getnetbyname, getnetent, setnetent — network database functions

SYNOPSIS

```
#include <netdb.h>

void endnetent(void);
struct netent *getnetbyaddr(uint32_t net, int type);
struct netent *getnetbyname(const char *name);
struct netent *getnetent(void);
void setnetent(int stayopen);
```

DESCRIPTION

These functions shall retrieve information about networks. This information is considered to be stored in a database that can be accessed sequentially or randomly. The implementation of this database is unspecified.

The *setnetent*( ) function shall open and rewind the database. If the *stayopen* argument is non-zero, the connection to the *net* database shall not be closed after each call to *getnetent*( ) (either directly, or indirectly through one of the other *getnet**( ) functions), and the implementation may maintain an open file descriptor to the database.

The *getnetent*( ) function shall read the next entry of the database, opening and closing a connection to the database as necessary.

The *getnetbyaddr*( ) function shall search the database from the beginning, and find the first entry for which the address family specified by *type* matches the *n_addrtype* member and the network number *net* matches the *n_net* member, opening and closing a connection to the database as necessary. The *net* argument shall be the network number in host byte order.

The *getnetbyname*( ) function shall search the database from the beginning and find the first entry for which the network name specified by *name* matches the *n_name* member, opening and closing a connection to the database as necessary.

The *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), functions shall each return a pointer to a **netent** structure, the members of which shall contain the fields of an entry in the network database.

The *endnetent*( ) function shall close the database, releasing any open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), shall return a pointer to a **netent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *getnetbyaddr*( ), *getnetbyname*( ), and *getnetent*( ), functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endprotoent, getprotobyname, getprotobynumber, getprotoent, setprotoent — network protocol database functions

SYNOPSIS

```
#include <netdb.h>

void endprotoent(void);
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
struct protoent *getprotoent(void);
void setprotoent(int stayopen);
```

DESCRIPTION

These functions shall retrieve information about protocols. This information is considered to be stored in a database that can be accessed sequentially or randomly. The implementation of this database is unspecified.

The *setprotoent*( ) function shall open a connection to the database, and set the next entry to the first entry. If the *stayopen* argument is non-zero, the connection to the network protocol database shall not be closed after each call to *getprotoent*( ) (either directly, or indirectly through one of the other *getproto**( ) functions), and the implementation may maintain an open file descriptor for the database.

The *getprotobyname*( ) function shall search the database from the beginning and find the first entry for which the protocol name specified by *name* matches the *p_name* member, opening and closing a connection to the database as necessary.

The *getprotobynumber*( ) function shall search the database from the beginning and find the first entry for which the protocol number specified by *proto* matches the *p_proto* member, opening and closing a connection to the database as necessary.

The *getprotoent*( ) function shall read the next entry of the database, opening and closing a connection to the database as necessary.

The *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ), functions shall each return a pointer to a **protoent** structure, the members of which shall contain the fields of an entry in the network protocol database.

The *endprotoent*( ) function shall close the connection to the database, releasing any open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ) return a pointer to a **protoent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *getprotobyname*( ), *getprotobynumber*( ), and *getprotoent*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endpwent, getpwent, setpwent — user database functions

SYNOPSIS

XSI
```
#include <pwd.h>

void endpwent(void);
struct passwd *getpwent(void);
void setpwent(void);
```

DESCRIPTION

These functions shall retrieve information about users.

The *getpwent*( ) function shall return a pointer to a structure containing the broken-out fields of an entry in the user database. Each entry in the user database contains a **passwd** structure. When first called, *getpwent*( ) shall return a pointer to a **passwd** structure containing the first entry in the user database. Thereafter, it shall return a pointer to a **passwd** structure containing the next entry in the user database. Successive calls can be used to search the entire user database.

If an end-of-file or an error is encountered on reading, *getpwent*( ) shall return a null pointer.

An implementation that provides extended security controls may impose further implementation-defined restrictions on accessing the user database. In particular, the system may deny the existence of some or all of the user database entries associated with users other than the caller.

The *setpwent*( ) function effectively rewinds the user database to allow repeated searches.

The *endpwent*( ) function may be called to close the user database when processing is complete.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *getpwent*( ) function shall return a null pointer on end-of-file or error.

ERRORS

The *getpwent*( ), *setpwent*( ), and *endpwent*( ) functions may fail if:

[EIO]	An I/O error has occurred.

In addition, *getpwent*( ) and *setpwent*( ) may fail if:

[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

The return value may point to a static area which is overwritten by a subsequent call to *getpwuid*( ), *getpwnam*( ), or *getpwent*( ).

EXAMPLES

Searching the User Database

The following example uses the *getpwent*( ) function to get successive entries in the user database, returning a pointer to a **passwd** structure that contains information about each user. The call to *endpwent*( ) closes the user database and cleans up.

```
#include <pwd.h>
...
struct passwd *p;
...
while ((p = getpwent ()) != NULL) {
...
}

endpwent();
...
```

APPLICATION USAGE

These functions are provided due to their historical usage. Applications should avoid dependencies on fields in the password database, whether the database is a single file, or where in the file system name space the database resides. Applications should use *getpwuid*( ) whenever possible because it avoids these dependencies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent( ), *getlogin*( ), *getpwnam*( ), *getpwuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pwd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

endservent, getservbyname, getservbyport, getservent, setservent — network services database functions

SYNOPSIS

```
#include <netdb.h>

void endservent(void);
struct servent *getservbyname(const char *name, const char *proto);
struct servent *getservbyport(int port, const char *proto);
struct servent *getservent(void);
void setservent(int stayopen);
```

DESCRIPTION

These functions shall retrieve information about network services. This information is considered to be stored in a database that can be accessed sequentially or randomly. The implementation of this database is unspecified.

The *setservent*( ) function shall open a connection to the database, and set the next entry to the first entry. If the *stayopen* argument is non-zero, the *net* database shall not be closed after each call to the *getservent*( ) function (either directly, or indirectly through one of the other *getserv**( ) functions), and the implementation may maintain an open file descriptor for the database.

The *getservent*( ) function shall read the next entry of the database, opening and closing a connection to the database as necessary.

The *getservbyname*( ) function shall search the database from the beginning and find the first entry for which the service name specified by *name* matches the *s_name* member and the protocol name specified by *proto* matches the *s_proto* member, opening and closing a connection to the database as necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched.

The *getservbyport*( ) function shall search the database from the beginning and find the first entry for which the port specified by *port* matches the *s_port* member and the protocol name specified by *proto* matches the *s_proto* member, opening and closing a connection to the database as necessary. If *proto* is a null pointer, any value of the *s_proto* member shall be matched. The *port* argument shall be in network byte order.

The *getservbyname*( ), *getservbyport*( ), and *getservent*( ) functions shall each return a pointer to a **servent** structure, the members of which shall contain the fields of an entry in the network services database.

The *endservent*( ) function shall close the database, releasing any open file descriptor.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getservbyname*( ), *getservbyport*( ), and *getservent*( ) return a pointer to a **servent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found. Otherwise, a null pointer is returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *port* argument of *getservbyport*( ) need not be compatible with the port values of all address families.

The *getservbyname*( ), *getservbyport*( ), and *getservent*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endhostent( ), *endprotoent*( ), *htonl*( ), *inet_addr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

endutxent, getutxent, getutxid, getutxline, pututxline, setutxent — user accounting database functions

SYNOPSIS

XSI
```
#include <utmpx.h>

void endutxent(void);
struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
```

DESCRIPTION

These functions shall provide access to the user accounting database.

The *getutxent*( ) function shall read the next entry from the user accounting database. If the database is not already open, it shall open it. If it reaches the end of the database, it shall fail.

The *getutxid*( ) function shall search forward from the current point in the database. If the *ut_type* value of the **utmpx** structure pointed to by *id* is BOOT_TIME, OLD_TIME, or NEW_TIME, then it shall stop when it finds an entry with a matching *ut_type* value. If the *ut_type* value is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then it shall stop when it finds an entry whose type is one of these four and whose *ut_id* member matches the *ut_id* member of the **utmpx** structure pointed to by *id*. If the end of the database is reached without a match, *getutxid*( ) shall fail.

The *getutxline*( ) function shall search forward from the current point in the database until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which also has a *ut_line* value matching that in the **utmpx** structure pointed to by *line*. If the end of the database is reached without a match, *getutxline*( ) shall fail.

The *getutxid*( ) or *getutxline*( ) function may cache data. For this reason, to use *getutxline*( ) to search for multiple occurrences, the application shall zero out the static data after each success, or *getutxline*( ) may return a pointer to the same **utmpx** structure.

There is one exception to the rule about clearing the structure before further reads are done. The implicit read done by *pututxline*( ) (if it finds that it is not already at the correct place in the user accounting database) shall not modify the static structure returned by *getutxent*( ), *getutxid*( ), or *getutxline*( ), if the application has modified this structure and passed the pointer back to *pututxline*( ).

For all entries that match a request, the *ut_type* member indicates the type of the entry. Other members of the entry shall contain meaningful data based on the value of the *ut_type* member as follows:

ut_type Member	Other Members with Meaningful Data
EMPTY	No others
BOOT_TIME	*ut_tv*
OLD_TIME	*ut_tv*
NEW_TIME	*ut_tv*
USER_PROCESS	*ut_id*, *ut_user* (login name of the user), *ut_line*, *ut_pid*, *ut_tv*
INIT_PROCESS	*ut_id*, *ut_pid*, *ut_tv*
LOGIN_PROCESS	*ut_id*, *ut_user* (implementation-defined name of the login process), *ut_pid*, *ut_tv*
DEAD_PROCESS	*ut_id*, *ut_pid*, *ut_tv*

An implementation that provides extended security controls may impose implementation-defined restrictions on accessing the user accounting database. In particular, the system may deny the existence of some or all of the user accounting database entries associated with users other than the caller.

If the process has appropriate privileges, the *pututxline*( ) function shall write out the structure into the user accounting database. It shall use *getutxid*( ) to search for a record that satisfies the request. If this search succeeds, then the entry shall be replaced. Otherwise, a new entry shall be made at the end of the user accounting database.

The *endutxent*( ) function shall close the user accounting database.

The *setutxent*( ) function shall reset the input to the beginning of the database. This should be done before each search for a new entry if it is desired that the entire database be examined.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getutxent*( ), *getutxid*( ), and *getutxline*( ) shall return a pointer to a **utmpx** structure containing a copy of the requested entry in the user accounting database. Otherwise, a null pointer shall be returned.

The return value may point to a static area which is overwritten by a subsequent call to *getutxid*( ) or *getutxline*( ).

Upon successful completion, *pututxline*( ) shall return a pointer to a **utmpx** structure containing a copy of the entry added to the user accounting database. Otherwise, a null pointer shall be returned.

The *endutxent*( ) and *setutxent*( ) functions shall not return a value.

ERRORS

No errors are defined for the *endutxent*( ), *getutxent*( ), *getutxid*( ), *getutxline*( ), and *setutxent*( ) functions.

The *pututxline*( ) function may fail if:

[EPERM]	The process does not have appropriate privileges.

EXAMPLES

None.

APPLICATION USAGE

The sizes of the arrays in the structure can be found using the *sizeof* operator.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<utmpx.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

A note indicating that these functions need not be reentrant is added to the DESCRIPTION.

Issue 6

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

NAME

environ — array of character pointers to the environment strings

SYNOPSIS

```
extern char **environ;
```

DESCRIPTION

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables and *exec*.

NAME

erand48 — generate uniformly distributed pseudo-random numbers

SYNOPSIS

XSI
```
#include <stdlib.h>

double erand48(unsigned short xsubi[3]);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

erf, erff, erfl — error functions

SYNOPSIS

```
#include <math.h>

double erf(double x);
float erff(float x);
long double erfl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the error function of their argument *x*, defined as:

$$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$$

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the value of the error function.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, ±0 shall be returned.

If *x* is ±Inf, ±1 shall be returned.

If *x* is subnormal, a range error may occur, and 2 * *x* /*sqrt*(π) should be returned.

ERRORS

These functions may fail if:

MX	Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

Underflow occurs when |*x*| < DBL_MIN * (*sqrt*(π) / 2).

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

erfc( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *erf*( ) function is no longer marked as an extension.

The *erfc*( ) function is now split out onto its own reference page.

The *erff*( ) and *erfl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

NAME

erfc, erfcf, erfcl — complementary error functions

SYNOPSIS

```
#include <math.h>

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the complementary error function 1.0 – *erf*(*x*).

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the value of the complementary error function.

If the correct value would cause underflow and is not representable, a range error may occur and MX either 0.0 (if representable), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, +1 shall be returned.

If *x* is –Inf, +2 shall be returned.

If *x* is +Inf, +0 shall be returned.

If the correct value would cause underflow and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions may fail if:

Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

The *erfc*( ) function is provided because of the extreme loss of relative accuracy if *erf*(*x*) is called for large *x* and the result subtracted from 1.0.

Note for IEEE Std 754-1985 **double**, 26.55 < *x* implies *erfc*(*x*) has underflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

erf( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *erfc*( ) function is no longer marked as an extension.

These functions are split out from the *erf*( ) reference page.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

erff, erfl — error functions

SYNOPSIS

```
#include <math.h>

float erff(float x);
long double erfl(long double x);
```

DESCRIPTION

Refer to *erf*( ).

NAME

errno — error return value

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

The lvalue *errno* is used by many functions to return error values.

Many functions provide an error number in *errno*. It has type **int** and is defined in **<errno.h>**. The value of *errno* shall be defined only after a call to a function for which it is explicitly stated to be set and until it is changed by the next function call or if the application assigns it a value. The value of *errno* should only be examined when it is indicated to be valid by a function's return value. Applications shall obtain the definition of *errno* by the inclusion of **<errno.h>**. No function in this volume of IEEE Std 1003.1-200x shall set *errno* to 0.

It is unspecified whether *errno* is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name *errno*, the behavior is undefined.

The symbolic values stored in *errno* are documented in the ERRORS sections on all relevant pages.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

Previously both POSIX and X/Open documents were more restrictive than the ISO C standard in that they required *errno* to be defined as an external variable, whereas the ISO C standard required only that *errno* be defined as a modifiable lvalue with type **int**.

A program that uses *errno* for error checking should set it to 0 before a function call, then inspect it before a subsequent function call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.3, the Base Definitions volume of IEEE Std 1003.1-200x, **<errno.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The following sentence is deleted from the DESCRIPTION: ‘‘The value of *errno* is 0 at program start-up, but is never set to 0 by any XSI function’’. The DESCRIPTION also no longer states that conforming implementations may support the declaration:

```
extern int errno;
```

Issue 6

Obsolescent text regarding defining *errno* as:

```
extern int errno
```

is removed.

Text regarding no function setting *errno* to zero to indicate an error is changed to no function shall set *errno* to zero. This is for alignment with the ISO / IEC 9899: 1999 standard.

NAME

environ, execl, execv, execle, execve, execlp, execvp — execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*,
    (char *)0, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

DESCRIPTION

The *exec* family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image.

When a C-language program is executed as a result of this call, it shall be entered as a C-language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a null pointer. The null pointer terminating the *argv* array is not counted in *argc*.

THR Conforming multi-threaded applications shall not use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable shall be considered a use of the *environ* variable to access that environment variable.

The arguments specified by a program with one of the *exec* functions shall be passed on to the new process image in the corresponding *main*( ) arguments.

The argument *path* points to a pathname that identifies the new process image file.

The argument *file* is used to construct a pathname that identifies the new process image file. If the *file* argument contains a slash character, the *file* argument shall be used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If this environment variable is not present, the results of the search are implementation-defined.

If the process image file is not a valid executable object, and the system does not recognize it as something that cannot be executed (and thus returns [EINVAL]), *execlp*( ) and *execvp*( ) shall use the contents of that file as standard input to a command interpreter conforming to *system*( ). In this case, the command interpreter becomes the new process image.

The arguments represented by *arg0*,... are pointers to null-terminated character strings. These strings shall constitute the argument list available to the new process image. The list is terminated by a null pointer. The argument *arg0* should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *argv* is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings shall constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process being started by one of the *exec* functions.

The argument *envp* is an array of character pointers to null-terminated strings. These strings shall constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

For those forms not containing an *envp* pointer (*execl*( ), *execv*( ), *execlp*( ), and *execvp*( )), the environment for the new process image shall be taken from the external variable *environ* in the calling process.

The number of bytes available for the new process' combined argument and environment lists is {ARG_MAX}. It is implementation-defined whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image shall remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged. For any file descriptor that is closed for this reason, file locks are removed as a result of the close as described in *close*( ). Locks that are not removed by closing of file descriptors remain unchanged.

Directory streams open in the calling process image shall be closed in the new process image.

The state of the floating-point environment in the new process image shall be set to the default.

XSI The state of conversion descriptors and message catalog descriptors in the new process image is undefined. For the new process image, the equivalent of:

```
setlocale(LC_ALL, "C")
```

shall be executed at start-up.

Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling process image shall be set to the default action in the new process image (see **<signal.h>**). If the SIGCHLD signal is set to be ignored by the calling process image, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new process image. XSI After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and the SA_ONSTACK flag shall be cleared for all signals.

After a successful call to any of the *exec* functions, any functions previously registered by *atexit*( ) are no longer registered.

XSI If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image shall be set to the user ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image shall remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by *setuid*( ).

XSI Any shared memory segments attached to the calling process image shall not be attached to the new process image.

SEM Any named semaphores open in the calling process shall be closed as if by appropriate calls to *sem_close*( ).

TYM Any blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*( ) was implicitly called to unmap them.

ML Memory locks established by the calling process via calls to *mlockall*( ) or *mlock*( ) shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to the *exec* function. If the *exec* function fails, the effect on memory locks is unspecified.

MF|SHM Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

PS For the SCHED_FIFO and SCHED_RR scheduling policies, the policy and priority settings shall not be changed by a call to an *exec* function. For other scheduling policies, the policy and priority settings on *exec* are implementation-defined.

TMR Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.

MSG All open message queue descriptors in the calling process shall be closed, as described in *mq_close*( ).

AIO Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the *exec* function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the *exec* function itself blocks awaiting such I/O completion. In no event, however, shall the new process image created by the *exec* function be affected by the presence of outstanding asynchronous I/O operations at the time the *exec* function is called. Whether any I/O is canceled, and which I/O may be canceled upon *exec*, is implementation-defined.

CPT The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being *exec*ed shall not be reinitialized or altered as a result of the *exec* function other than to reflect the time spent by the process executing the *exec* function itself.

TCT The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.

TRC If the calling process is being traced, the new process image shall continue to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the *posix_trace_eventid_open*( ) or the *posix_trace_trid_eventid_open*( ) functions in the calling process image.

If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the *posix_trace_shutdown*( ) function.

The new process shall inherit at least the following attributes from the calling process image:

- XSI Nice value (see *nice*( ))
- XSI *semadj* values (see *semop*( ))

- Process ID
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- Time left until an alarm clock signal (see *alarm*( ))
- Current working directory
- Root directory
- File mode creation mask (see *umask*( ))
- XSI File size limit (see *ulimit*( ))
- Process signal mask (see *sigprocmask*( ))
- Pending signal (see *sigpending*( ))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*( ))
- XSI Resource limits
- XSI Controlling terminal
- XSI Interval timers

All other process attributes defined in this volume of IEEE Std 1003.1-200x shall be the same in the new and old process images. The inheritance of process attributes not defined by this volume of IEEE Std 1003.1-200x is implementation-defined.

A call to any *exec* function from a process with more than one thread shall result in all threads being terminated and the new executable image being loaded and executed. No destructor functions shall be called.

Upon successful completion, the *exec* functions shall mark for update the *st_atime* field of the file. If an *exec* function failed but was able to locate the *process image file*, whether the *st_atime* field is marked for update is unspecified. Should the *exec* function succeed, the process image file shall be considered to have been opened with *open*( ). The corresponding *close*( ) shall be considered to occur at a time after this open, but before process termination or successful completion of a subsequent call to one of the *exec* functions, *posix_spawn*( ), or *posix_spawnp*( ). The *argv*[ ] and *envp*[ ] arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the *exec* functions, except as a consequence of replacing the process image.

XSI The saved resource limits in the new process image are set to be a copy of the process' corresponding hard and soft limits.

RETURN VALUE

If one of the *exec* functions returns to the calling process image, an error has occurred; the return value shall be −1, and *errno* shall be set to indicate the error.

ERRORS

The *exec* functions shall fail if:

[E2BIG]	The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.
[EACCES]	Search permission is denied for a directory listed in the new process image file's path prefix, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.
[EINVAL]	The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* or *file* argument.
[ENAMETOOLONG]	The length of the *path* or *file* arguments exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* or *file* does not name an existing file or *path* or *file* is an empty string.
[ENOTDIR]	A component of the new process image file's path prefix is not a directory.

The *exec* functions, except for *execlp*( ) and *execvp*( ), shall fail if:

[ENOEXEC]	The new process image file has the appropriate access permission but has an unrecognized format.

The *exec* functions may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* or *file* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
[ENOMEM]	The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.
[ETXTBSY]	The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

EXAMPLES

Using execl( )

The following example executes the *ls* command, specifying the pathname of the executable (**/bin/ls**) and using arguments supplied directly to the command to produce single-column output.

```
#include <unistd.h>

int ret;
...
ret = execl ("/bin/ls", "ls", "-1", (char *)0);
```

Using execle()

The following example is similar to **Using execl()** (on page 758). In addition, it specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

Using execlp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable.

```
#include <unistd.h>

int ret;
...
ret = execlp ("ls", "ls", "-l", (char *)0);
```

Using execv()

The following example passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execv ("/bin/ls", cmd);
```

Using execve()

The following example passes arguments to the *ls* command in the *cmd* array, and specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execve ("/bin/ls", cmd, env);
```

Using execvp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable, and passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
```

```
        ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, conforming applications should not rely on their use and should close them prior to calling one of the *exec* functions. |

Applications that require other than the default POSIX locale should call *setlocale*( ) with the appropriate parameters to establish the locale of the new process.

The *environ* array should not be accessed directly by the application.

RATIONALE

Early proposals required that the value of *argc* passed to *main*( ) be ''one or greater''. This was driven by the same requirement in drafts of the ISO C standard. In fact, historical implementations have passed a value of zero when no arguments are supplied to the caller of the *exec* functions. This requirement was removed from the ISO C standard and subsequently removed from this volume of IEEE Std 1003.1-200x as well. The wording, in particular the use of the word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument to the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an application. In fact, this is good practice, since many existing applications reference *argv*[0] without first checking the value of *argc*.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename associated with the process being started. Although some existing applications pass a pathname rather than a filename in some circumstances, a filename is more generally useful, since the common usage of *argv*[0] is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example, many implementations of the *login* utility use a convention of prefixing a hyphen ('−') to the actual filename, which indicates to the command interpreter being invoked that it is a ''login shell''. |

Some implementations can *exec* shell scripts. |

One common historical implementation is that the *execl*( ), *execv*( ), *execle*( ), and *execve*( ) functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the *execlp*( ) and *execvp*( ) functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to interpret such files. These implementations of *execvp*( ) and *execlp*( ) only give the [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file. Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp*( ) or *execvp*( ), although implementations can still give it.

Another way that some historical implementations handle shell scripts is by recognizing the first two bytes of the file as the character string `"#!"` and using the remainder of the first line of the file as the name of the command interpreter to execute.

Some implementations provide a third argument to *main*( ) called *envp*. This is defined as a pointer to the environment. The ISO C standard specifies invoking *main*( ) with two arguments, so implementations must support applications written this way. Since this volume of IEEE Std 1003.1-200x defines the global variable *environ*, which is also provided by historical implementations and can be used anywhere that *envp* could be used, there is no functional need for the *envp* argument. Applications should use the *getenv*( ) function rather than accessing the environment directly via either *envp* or *environ*. Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting *envp* as an optional third argument.

This volume of IEEE Std 1003.1-200x specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the process signal mask be unchanged across an *exec*. This is consistent with historical implementations, and it permits some useful functionality, such as the *nohup* command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the ISO C standard, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across *exec*s without explicit reason to do so, and especially not to block signals across *exec*s of arbitrary (not closely co-operating) programs.

The *exec* functions always save the value of the effective user ID and effective group ID of the process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of the process image file is set.

The statement about *argv*[ ] and *envp*[ ] being constants is included to make explicit to future writers of language bindings that these objects are completely constant. Due to a limitation of the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of *const–qualification* for the *argv*[ ] and *envp*[ ] parameters for the *exec* functions may seem to be the natural choice, given that these functions do not modify either the array of pointers or the characters to which the function points, but this would disallow existing correct code. Instead, only the array of pointers is noted as constant. The table of assignment compatibility for *dst*=*src*, derived from the ISO C standard summarizes the compatibility:

dst:	**char *[ ]**	**const char *[ ]**	**char *const[ ]**	**const char *const[ ]**
src:				
char *[ ]	VALID	—	VALID	—
const char *[ ]	—	VALID	—	VALID
char * const [ ]	—	—	VALID	—
const char *const[ ]	—	—	—	VALID

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of IEEE Std 1003.1-200x do not conflict on the use of *environ*, but some historical implementations of *environ* may cause a conflict. As long as *environ* is treated in the same way as an entry point (for example, *fork*( )), it conforms to both standards. A library can contain *fork*( ), but if there is a user-provided *fork*( ), that *fork*( ) is given precedence and no problem ensues. The situation is similar for *environ*: the definition in this volume of IEEE Std 1003.1-200x is to be used if there is no user-provided *environ* to take precedence. At least three implementations are known to exist that solve this problem.

[E2BIG]	The limit {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.
[EFAULT]	Some historical systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.
[EINVAL]	This error condition was added to IEEE Std 1003.1-200x to allow an implementation to detect executable files generated for different architectures, and indicate this situation to the application. Historical implementations of shells, *execvp*( ), and *execlp*( ) that encounter an [ENOEXEC] error will execute a shell on the assumption that the file is a shell script. This will not produce the desired effect when the file is a valid executable for a different

architecture. An implementation may now choose to avoid this problem by returning [EINVAL] when a valid executable for a different architecture is encountered. Some historical implementations return [EINVAL] to indicate that the *path* argument contains a character with the high order bit set. The standard developers chose to deviate from historical practice for the following reasons:

1. The new utilization of [EINVAL] will provide some measure of utility to the user community.
2. Historical use of [EINVAL] is not acceptable in an internationalized operating environment.

[ENAMETOOLONG]
Since the file pathname may be constructed by taking elements in the *PATH* variable and putting them together with the filename, the [ENAMETOOLONG] error condition could also be reached this way.

[ETXTBSY] System V returns this error when the executable file is currently open for writing by some process. This volume of IEEE Std 1003.1-200x neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by this volume of IEEE Std 1003.1-200x, but implementations may have a window between the call to *exec* and the time that a signal could cause one of the *exec* calls to return with [EINTR].

An explicit statement regarding the floating-point environment (as defined in the **<fenv.h>** header) was added to make it clear that the floating-point environment is set to its default when a call to one of the *exec* functions succeeds. The requirements for inheritance or setting to the default for other process and thread start-up functions is covered by more generic statements in their descriptions and can be summarized as follows:

posix_spawn( )	**Set to default.**
fork( )	Inherit.
pthread_create( )	Inherit.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *atexit*( ), *chmod*( ), *close*( ), *exit*( ), *fcntl*( ), *fork*( ), *fstatvfs*( ), *getenv*( ), *getitimer*( ), *getrlimit*( ), *mmap*( ), *nice*( ), *posix_spawn*( ), *posix_trace_eventid_open*( ), *posix_trace_shutdown*( ), *posix_trace_trid_eventid_open*( ), *putenv*( ), *semop*( ), *setlocale*( ), *shmat*( ), *sigaction*( ), *sigaltstack*( ), *sigpending*( ), *sigprocmask*( ), *system*( ), *times*( ), *ulimit*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.
- In this issue, _POSIX_SAVED_IDS is mandated, thus the effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the *setuid*( ) function.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [EINVAL] mandatory error condition is added.
- The [ELOOP] optional error condition is added.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #132 is applied.

The DESCRIPTION is updated to make it explicit that the floating-point environment in the new process image is set to the default.

NAME

exit, _Exit, _exit — terminate a process

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);
void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

DESCRIPTION

The functionality described on this reference page for the *exit*( ) and *_Exit*( ) functions is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard are unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The value of *status* may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits (that is, *status* & 0377) shall be available to a waiting parent process.

The *exit*( ) function shall first call all functions registered by *atexit*( ), in the reverse order of their registration, except that a function is called after any previously registered functions that had already been called at the time it was registered. Each function is called as many times as it was registered. If, during the call to any such function, a call to the *longjmp*( ) function is made that would terminate the call to the registered function, the behavior is undefined.

If a function registered by a call to *atexit*( ) fails to return, the remaining registered functions shall not be called and the rest of the *exit*( ) processing shall not be completed. If *exit*( ) is called more than once, the behavior is undefined.

The *exit*( ) function shall then flush all open streams with unwritten buffered data, close all open streams, and remove all files created by *tmpfile*( ). Finally, control shall be terminated with the consequences described below.

The *_Exit*( ) and *_exit*( ) functions shall be functionally equivalent.

The *_Exit*( ) and *_exit*( ) functions shall not call functions registered with *atexit*( ) nor any registered signal handlers. Whether open streams are flushed or closed, or temporary files are removed is implementation-defined. Finally, the calling process is terminated with the consequences described below.

These functions shall terminate the calling process with the following consequences:

Note: These consequences are all extensions to the ISO C standard and are not further CX shaded. However, XSI extensions are shaded.

- All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process shall be closed.
- If the parent process of the calling process is executing a *wait*( ) or *waitpid*( ), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it shall be notified of the calling process' termination and the low-order eight bits (that is, bits 0377) of *status* are made available to it. If the parent is not waiting, the child's status shall be made available to it when the parent subsequently executes *wait*( ) or *waitpid*( ).

  The semantics of the *waitid*( ) function shall be equivalent to *wait*( ).
- If the parent process of the calling process is not executing a *wait*( ) or *waitpid*( ), and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, the calling process shall be transformed into a *zombie process*. A *zombie process* is an inactive process and it shall be

deleted at some later time when its parent process executes *wait*( ) or *waitpid*( ).

XSI The semantics of the *waitid*( ) function shall be equivalent to *wait*( ).

- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly terminates children in some circumstances.
- Either:

  If the implementation supports the SIGCHLD signal, a SIGCHLD shall be sent to the parent process.

  Or:

  XSI If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status shall be discarded, and the lifetime of the calling process shall end immediately. If SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal is sent to the parent process.
- The parent process ID of all of the calling process' existing child processes and zombie processes shall be set to the process ID of an implementation-defined system process. That is, these processes shall be inherited by a special system process.
- XSI Each attached shared-memory segment is detached and the value of *shm_nattch* (see *shmget*( )) in the data structure associated with its shared memory ID shall be decremented by 1.
- XSI For each semaphore for which the calling process has set a *semadj* value (see *semop*( )), that value shall be added to the *semval* of the specified semaphore.
- If the process is a controlling process, the SIGHUP signal shall be sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session shall be disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal shall be sent to each process in the newly-orphaned process group.
- SEM All open named semaphores in the calling process shall be closed as if by appropriate calls to *sem_close*( ).
- ML Any memory locks established by the process via calls to *mlockall*( ) or *mlock*( ) shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by the call by this process to *_Exit*( ) or *_exit*( ).
- MF|SHM Memory mappings created in the process shall be unmapped before the process is destroyed.
- TYM Any blocks of typed memory that were mapped in the calling process shall be unmapped, as if *munmap*( ) was implicitly called to unmap them.
- MSG All open message queue descriptors in the calling process shall be closed as if by appropriate calls to *mq_close*( ).
- AIO Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled shall complete as if the *_Exit*( ) or *_exit*( ) operation had not yet occurred, but any associated signal notifications shall be suppressed.

The *_Exit*( ) or *_exit*( ) operation may block awaiting such I/O completion. Whether any I/O is canceled, and which I/O may be canceled upon *_Exit*( ) or *_exit*( ), is implementation-defined.

- Threads terminated by a call to *_Exit*( ) or *_exit*( ) shall not invoke their cancelation cleanup handlers or per-thread data destructors.
- TRC If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described by the *posix_trace_shutdown*( ) function, and any process' mapping of trace event names to trace event type identifiers built for these trace streams may be deallocated.

RETURN VALUE

These functions do not return.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Normally applications should use *exit*( ) rather than *_Exit*( ) or *_exit*( ).

RATIONALE

Process Termination

Early proposals drew a distinction between normal and abnormal process termination. Abnormal termination was caused only by certain signals and resulted in implementation-defined ‘‘actions’’, as discussed below. Subsequent proposals distinguished three types of termination: *normal termination* (as in the current specification), *simple abnormal termination*, and *abnormal termination with actions*. Again the distinction between the two types of abnormal termination was that they were caused by different signals and that implementation-defined actions would result in the latter case. Given that these actions were completely implementation-defined, the early proposals were only saying when the actions could occur and how their occurrence could be detected, but not what they were. This was of little or no use to conforming applications, and thus the distinction is not made in this volume of IEEE Std 1003.1-200x.

The implementation-defined actions usually include, in most historical implementations, the creation of a file named **core** in the current working directory of the process. This file contains an image of the memory of the process, together with descriptive information about the process, perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

There is a potential security problem in creating a **core** file if the process was set-user-ID and the current user is not the owner of the program, if the process was set-group-ID and none of the user's groups match the group of the program, or if the user does not have permission to write in the current directory. In this situation, an implementation either should not create a **core** file or should make it unreadable by the user.

Despite the silence of this volume of IEEE Std 1003.1-200x on this feature, applications are advised not to create files named **core** because of potential conflicts in many implementations. Some historical implementations use a different name than **core** for the file, such as by appending the process ID to the filename.

Terminating a Process

It is important that the consequences of process termination as described occur regardless of whether the process called *_exit*( ) (perhaps indirectly through *exit*( )) or instead was terminated due to a signal or for some other reason. Note that in the specific case of *exit*( ) this means that the *status* argument to *exit*( ) is treated in the same way as the *status* argument to *_exit*( ).

A language other than C may have other termination primitives than the C-language *exit*( ) function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of *_exit*( ) as described. Implementations in languages other than C are outside the scope of the present version of this volume of IEEE Std 1003.1-200x, however.

As required by the ISO C standard, using **return** from *main*( ) has the same behavior (other than with respect to language scope issues) as calling *exit*( ) with the returned value. Reaching the end of the *main*( ) function has the same behavior as calling *exit*(0).

A value of zero (or EXIT_SUCCESS, which is required to be zero) for the argument *status* conventionally indicates successful termination. This corresponds to the specification for *exit*( ) in the ISO C standard. The convention is followed by utilities such as *make* and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call *exit*(0) or *_exit*(0) when they terminate unsuccessfully; for example, in signal-catching functions.

Historically, the implementation-defined process that inherits children whose parents have terminated without waiting on them is called *init* and has a process ID of 1.

The sending of a SIGHUP to the foreground process group when a controlling process terminates corresponds to somewhat different historical implementations. In System V, the kernel sends a SIGHUP on termination of (essentially) a controlling process. In 4.2 BSD, the kernel does not send SIGHUP in a case like this, but the termination of a controlling process is usually noticed by a system daemon, which arranges to send a SIGHUP to the foreground process group with the *vhangup*( ) function. However, in 4.2 BSD, due to the behavior of the shells that support job control, the controlling process is usually a shell with no other processes in its process group. Thus, a change to make *_exit*( ) behave this way in such systems should not cause problems with existing applications.

The termination of a process may cause a process group to become orphaned in either of two ways. The connection of a process group to its parent(s) outside of the group depends on both the parents and their children. Thus, a process group may be orphaned by the termination of the last connecting parent process outside of the group or by the termination of the last direct descendant of the parent process(es). In either case, if the termination of a process causes a process group to become orphaned, processes within the group are disconnected from their job control shell, which no longer has any information on the existence of the process group. Stopped processes within the group would languish forever. In order to avoid this problem, newly orphaned process groups that contain stopped processes are sent a SIGHUP signal and a SIGCONT signal to indicate that they have been disconnected from their session. The SIGHUP signal causes the process group members to terminate unless they are catching or ignoring SIGHUP. Under most circumstances, all of the members of the process group are stopped if any of them are stopped.

The action of sending a SIGHUP and a SIGCONT signal to members of a newly orphaned process group is similar to the action of 4.2 BSD, which sends SIGHUP and SIGCONT to each stopped child of an exiting process. If such children exit in response to the SIGHUP, any additional descendants receive similar treatment at that time. In this volume of IEEE Std 1003.1-200x, the signals are sent to the entire process group at the same time. Also, in this volume of IEEE Std 1003.1-200x, but not in 4.2 BSD, stopped processes may be orphaned, but may be members of a process group that is not orphaned; therefore, the action taken at *_exit*( ) must consider processes other than child processes.

It is possible for a process group to be orphaned by a call to *setpgid*( ) or *setsid*( ), as well as by process termination. This volume of IEEE Std 1003.1-200x does not require sending SIGHUP and SIGCONT in those cases, because, unlike process termination, those cases are not caused accidentally by applications that are unaware of job control. An implementation can choose to send SIGHUP and SIGCONT in those cases as an extension; such an extension must be documented as required in **<signal.h>**.

The ISO / IEC 9899: 1999 standard adds the *_Exit*( ) function that results in immediate program termination without triggering signals or *atexit*( )-registered functions. In IEEE Std 1003.1-200x, this is equivalent to the *_exit*( ) function.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *close*( ), *fclose*( ), *longjmp*( ), *posix_trace_shutdown*( ), *posix_trace_trid_eventid_open*( ), *semop*( ), *shmget*( ), *sigaction*( ), *wait*( ), *waitid*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Interactions with the SA_NOCLDWAIT flag and SIGCHLD signal are further clarified.

The values of *status* from *exit*( ) are better described.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *_Exit*( ) function is included.
- The DESCRIPTION is updated.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

References to the *wait3*( ) function are removed.

NAME

exp, expf, expl — exponential function

SYNOPSIS

```
#include <math.h>

double exp(double x);
float expf(float x);
long double expl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the base-*e* exponential of *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the exponential value of *x*.

If the correct value would cause overflow, a range error shall occur and *exp*( ), *expf*( ), and *expl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, 1 shall be returned.

If *x* is –Inf, +0 shall be returned.

If *x* is +Inf, *x* shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

Note that for IEEE Std 754-1985 **double**, $709.8 < x$ implies *exp*(x) has overflowed. The value $x < -708.4$ implies *exp*(x) has underflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *expf*( ) and *expl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

exp2, exp2f, exp2l — exponential base 2 functions

SYNOPSIS

```
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the base-2 exponential of *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return 2^x.

If the correct value would cause overflow, a range error shall occur and *exp2*( ), *exp2f*( ), and *exp2l*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, 1 shall be returned.

If *x* is –Inf, +0 shall be returned.

If *x* is +Inf, *x* shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

Range Error	The result overflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

For IEEE Std 754-1985 **double**, 1024 <= *x* implies *exp2*(*x*) has overflowed. The value *x* < −1022 implies *exp*(*x*) has underflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *log*( ), the Base Definitions volume of | IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, | **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

expm1, expm1f, expm1l — compute exponential functions

SYNOPSIS

```
#include <math.h>

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute $e^{x}-1.0$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions return $e^{x}-1.0$.

If the correct value would cause overflow, a range error shall occur and *expm1*( ), *expm1f*( ), and *expm1l*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, ±0 shall be returned.

If *x* is –Inf, –1 shall be returned.

If *x* is +Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

Range Error	The result overflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

MX Range Error	The value of *x* is subnormal. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

The value of *expm1*(x) may be more accurate than *exp*(x)–1.0 for small values of x.

The *expm1*( ) and *log1p*( ) functions are useful for financial calculations of $((1+x)^n-1)/x$, namely:

```
expm1(n * log1p(x))/x
```

when x is very small (for example, when calculating small daily interest rates). These functions also simplify writing accurate inverse hyperbolic functions.

For IEEE Std 754-1985 **double**, $709.8 < x$ implies *expm1*(x) has overflowed.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *ilogb*( ), *log1p*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *expm1f*( ) and *expm1l*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The *expm1*( ) function is no longer marked as an extension. |

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard. |

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

fabs, fabsf, fabsl — absolute value function

SYNOPSIS

```
#include <math.h>

double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the absolute value of their argument *x*, |*x*|.

RETURN VALUE

Upon successful completion, these functions shall return the absolute value of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, +0 shall be returned.

If *x* is ±Inf, +Inf shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isnan( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *fabsf*( ) and *fabsl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

fattach — attach a STREAMS-based file descriptor to a file in the file system name space (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>

int fattach(int fildes, const char *path);
```

DESCRIPTION

The *fattach*( ) function shall attach a STREAMS-based file descriptor to a file, effectively associating a pathname with *fildes*. The application shall ensure that the *fildes* argument is a valid open file descriptor associated with a STREAMS file. The *path* argument points to a pathname of an existing file. The application shall have the appropriate privileges, or is the owner of the file named by *path* and has write permission. A successful call to *fattach*( ) shall cause all pathnames that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file shall be initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If any attributes of the named STREAMS file are subsequently changed (for example, by *chmod*( )), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes* refers shall be affected.

File descriptors referring to the underlying file, opened prior to an *fattach*( ) call, shall continue to refer to the underlying file.

RETURN VALUE

Upon successful completion, *fattach*( ) shall return 0. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fattach*( ) function shall fail if:

[EACCES]	Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permissions on the file named by *path*.
[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EBUSY]	The file named by *path* is currently a mount point or has a STREAMS file attached to it.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The size of *path* exceeds {PATH_MAX} or a component of *path* is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID of the process is not the owner of the file named by *path* and the process does not have appropriate privilege.

The *fattach*( ) function may fail if:

[EINVAL] The *fildes* argument does not refer to a STREAMS file.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

[EXDEV] A link to a file on another file system was attempted.

EXAMPLES

Attaching a File Descriptor to a File

In the following example, *fd* refers to an open STREAMS file. The call to *fattach*( ) associates this STREAM with the file **/tmp/named-STREAM**, such that any future calls to open **/tmp/named-STREAM**, prior to breaking the attachment via a call to *fdetach*( ), will instead create a new file handle referring to the STREAMS file associated with *fd*.

```
#include <stropts.h>
...
    int fd;
    char *filename = "/tmp/named-STREAM";
    int ret;

    ret = fattach(fd, filename);
```

APPLICATION USAGE

The *fattach*( ) function behaves similarly to the traditional *mount*( ) function in the way a file is temporarily replaced by the root directory of the mounted file system. In the case of *fattach*( ), the replaced file need not be a directory and the replacing file is a STREAMS file.

RATIONALE

The file attributes of a file which has been the subject of an *fattach*( ) call are specifically set because of an artefact of the original implementation. The internal mechanism was the same as for the *mount*( ) function. Since *mount*( ) is typically only applied to directories, the effects when applied to a regular file are a little surprising, especially as regards the link count which rigidly remains one, even if there were several links originally and despite the fact that all original links refer to the STREAM as long as the *fattach*( ) remains in effect. |

FUTURE DIRECTIONS

None.

SEE ALSO

fdetach( ), *isastream*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The [EXDEV] error is added to the list of optional errors in the ERRORS section.

Issue 6

This function is marked as part of the XSI STREAMS Option Group.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

fchdir — change working directory

SYNOPSIS

XSI

```
#include <unistd.h>

int fchdir(int fildes);
```

DESCRIPTION

The *fchdir*( ) function shall be equivalent to *chdir*( ) except that the directory that is to be the new current working directory is specified by the file descriptor *fildes*.

A conforming application can obtain a file descriptor for a file of type directory using *open*( ), provided that the file status flags and access modes do not contain O_WRONLY or O_RDWR.

RETURN VALUE

Upon successful completion, *fchdir*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error. On failure the current working directory shall remain unchanged.

ERRORS

The *fchdir*( ) function shall fail if:

[EACCES]	Search permission is denied for the directory referenced by *fildes*.
[EBADF]	The *fildes* argument is not an open file descriptor.
[ENOTDIR]	The open file descriptor *fildes* does not refer to a directory.

The *fchdir*( ) may fail if:

[EINTR]	A signal was caught during the execution of *fchdir*( ).
[EIO]	An I/O error occurred while reading from or writing to the file system.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chdir( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

fchmod — change mode of a file

SYNOPSIS

```
#include <sys/stat.h>

int fchmod(int fildes, mode_t mode);
```

DESCRIPTION

The *fchmod*( ) function shall be equivalent to *chmod*( ) except that the file whose permissions are changed is specified by the file descriptor *fildes*.

SHM If *fildes* references a shared memory object, the *fchmod*( ) function need only affect the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits.

TYM If *fildes* references a typed memory object, the behavior of *fchmod*( ) is unspecified.

If *fildes* refers to a socket, the behavior of *fchmod*( ) is unspecified.

XSR If *fildes* refers to a STREAM (which is *fattach*( )-ed into the file system name space) the call returns successfully, doing nothing.

RETURN VALUE

Upon successful completion, *fchmod*( ) shall return 0. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *fchmod*( ) function shall fail if:

[EBADF]	The *fildes* argument is not an open file descriptor.
[EPERM]	The effective user ID does not match the owner of the file and the process does not have appropriate privilege.
[EROFS]	The file referred to by *fildes* resides on a read-only file system.

The *fchmod*( ) function may fail if:

XSI	[EINTR]	The *fchmod*( ) function was interrupted by a signal.
XSI	[EINVAL]	The value of the *mode* argument is invalid.
	[EINVAL]	The *fildes* argument refers to a pipe and the implementation disallows execution of *fchmod*( ) on a pipe.

EXAMPLES

Changing the Current Permissions for a File

The following example shows how to change the permissions for a file named **/home/cnd/mod1** so that the owner and group have read/write/execute permissions, but the world only has read/write permissions.

```
#include <sys/stat.h>
#include <fcntl.h>

mode_t mode;
int    fildes;
...
fildes = open("/home/cnd/mod1", O_RDWR);
fchmod(fildes, S_IRWXU | S_IRWXG | S_IROTH | S_IWOTH);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *chown*( ), *creat*( ), *fcntl*( ), *fstatvfs*( ), *mknod*( ), *open*( ), *read*( ), *stat*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE and aligned with *fchmod*( ) in the POSIX Realtime Extension. Specifically, the second paragraph of the DESCRIPTION is added and a second instance of [EINVAL] is defined in the list of optional errors.

Issue 6

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by stating that *fchmod*( ) behavior is unspecified for typed memory objects.

NAME

fchown — change owner and group of a file

SYNOPSIS

```
#include <unistd.h>

int fchown(int fildes, uid_t owner, gid_t group);
```

DESCRIPTION

The *fchown*( ) function shall be equivalent to *chown*( ) except that the file whose owner and group are changed is specified by the file descriptor *fildes*.

RETURN VALUE

Upon successful completion, *fchown*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *fchown*( ) function shall fail if:

[EBADF]	The *fildes* argument is not an open file descriptor.
[EPERM]	The effective user ID does not match the owner of the file or the process does not have appropriate privilege and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.
[EROFS]	The file referred to by *fildes* resides on a read-only file system.

The *fchown*( ) function may fail if:

[EINVAL]	The owner or group ID is not a value supported by the implementation. The *fildes* argument refers to a pipe or socket or an *fattach*( )-ed STREAM and the implementation disallows execution of *fchown*( ) on a pipe. (XSR)
[EIO]	A physical I/O error has occurred.
[EINTR]	The *fchown*( ) function was interrupted by a signal which was caught.

EXAMPLES

Changing the Current Owner of a File

The following example shows how to change the owner of a file named **/home/cnd/mod1** to ‘‘jones’’ and the group to ‘‘cnd’’.

The numeric value for the user ID is obtained by extracting the user ID from the user database entry associated with ‘‘jones’’. Similarly, the numeric value for the group ID is obtained by extracting the group ID from the group database entry associated with ‘‘cnd’’. This example assumes the calling program has appropriate privileges.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <pwd.h>
#include <grp.h>

struct passwd *pwd;
struct group  *grp;
int            fildes;
...
fildes = open("/home/cnd/mod1", O_RDWR);
pwd = getpwnam("jones");
```

```
    grp = getgrnam("cnd");
    fchown(fildes, pwd->pw_uid, grp->gr_gid);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that a call to *fchown*( ) may not be allowed on a pipe.

The *fchown*( ) function is now defined as mandatory.

NAME

fclose — close a stream

SYNOPSIS

```
#include <stdio.h>

int fclose(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fclose*( ) function shall cause the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream shall be written to the file; any unread buffered data shall be discarded. Whether or not the call succeeds, the stream shall be disassociated from the file and any buffer set by the *setbuf*( ) or *setvbuf*( ) function shall be disassociated from the stream. If the associated buffer was automatically allocated, it shall be deallocated.

CX The *fclose*( ) function shall mark for update the *st_ctime* and *st_mtime* fields of the underlying file, if the stream was writable, and if buffered data remains that has not yet been written to the file. The *fclose*( ) function shall perform the equivalent of a *close*( ) on the file descriptor that is associated with the stream pointed to by *stream*.

After the call to *fclose*( ), any use of *stream* results in undefined behavior.

RETURN VALUE

CX Upon successful completion, *fclose*( ) shall return 0; otherwise, it shall return EOF and set *errno* to indicate the error.

ERRORS

The *fclose*( ) function shall fail if:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
CX	[EBADF]	The file descriptor underlying stream is not valid.
CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.
CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
CX	[EINTR]	The *fclose*( ) function was interrupted by a signal.
CX	[EIO]	The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
CX	[ENOSPC]	There was no free space remaining on the device containing the file.
CX	[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The *fclose*( ) function may fail if:

CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *fopen*( ), *getrlimit*( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.

The DESCRIPTION is updated to note that the stream and any buffer are disassociated whether or not the call succeeds. This is for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fcntl — file control

SYNOPSIS

OH
```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fildes, int cmd, ...);
```

DESCRIPTION

The *fcntl*( ) function shall perform the operations described below on open files. The *fildes* argument is a file descriptor.

The available values for *cmd* are defined in **<fcntl.h>** and are as follows:

F_DUPFD	Return a new file descriptor which shall be the lowest numbered available (that is, not already open) file descriptor greater than or equal to the third argument, *arg*, taken as an integer of type **int**. The new file descriptor shall refer to the same open file description as the original file descriptor, and shall share any locks. The FD_CLOEXEC flag associated with the new file descriptor shall be cleared to keep the file open across calls to one of the *exec* functions.
F_GETFD	Get the file descriptor flags defined in **<fcntl.h>** that are associated with the file descriptor *fildes*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.
F_SETFD	Set the file descriptor flags defined in **<fcntl.h>**, that are associated with *fildes*, to the third argument, *arg*, taken as type **int**. If the FD_CLOEXEC flag in the third argument is 0, the file shall remain open across the *exec* functions; otherwise, the file shall be closed upon successful execution of one of the *exec* functions.
F_GETFL	Get the file status flags and file access modes, defined in **<fcntl.h>**, for the file description associated with *fildes*. The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in **<fcntl.h>**. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions.
F_SETFL	Set the file status flags, defined in **<fcntl.h>**, for the file description associated with *fildes* from the corresponding bits in the third argument, *arg*, taken as type **int**. Bits corresponding to the file access mode and the file creation flags, as defined in **<fcntl.h>**, that are set in *arg* shall be ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.
F_GETOWN	If *fildes* refers to a socket, get the process or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values indicate a process ID; negative values, other than −1, indicate a process group ID. If *fildes* does not refer to a socket, the results are unspecified.
F_SETOWN	If *fildes* refers to a socket, set the process or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, *arg*, taken as type **int**. Positive values indicate a process ID; negative values, other than −1, indicate a process group ID. If *fildes* does not refer to a socket, the results are unspecified.

The following values for *cmd* are available for advisory record locking. Record locking shall be supported for regular files, and may be supported for other files.

F_GETLK	Get the first lock which blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. The information retrieved shall overwrite the information passed to *fcntl*( ) in the structure **flock**. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be set to F_UNLCK.
F_SETLK	Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type **struct flock**, defined in **<fcntl.h>**. F_SETLK can establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK), as well as to remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK, and F_UNLCK are defined in **<fcntl.h>**. If a shared or exclusive lock cannot be set, *fcntl*( ) shall return immediately with a return value of –1.
F_SETLKW	This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied. If a signal that is to be caught is received while *fcntl*( ) is waiting for a region, *fcntl*( ) shall be interrupted. Upon return from the signal handler, *fcntl*( ) shall return –1 with *errno* set to [EINTR], and the lock operation shall not be done.

Additional implementation-defined values for *cmd* may be defined in **<fcntl.h>**. Their names shall start with F_.

When a shared lock is set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The structure **flock** describes the type (*l_type*), starting offset (*l_whence*), relative offset (*l_start*), size (*l_len*), and process ID (*l_pid*) of the segment of the file to be affected.

The value of *l_whence* is SEEK_SET, SEEK_CUR, or SEEK_END, to indicate that the relative offset *l_start* bytes shall be measured from the start of the file, current position, or end of the file, respectively. The value of *l_len* is the number of consecutive bytes to be locked. The value of *l_len* may be negative (where the definition of **off_t** permits negative values of *l_len*). The *l_pid* field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, when a blocking lock is found, the values returned in the **flock** structure shall be as follows:

l_type	Type of blocking lock found.
l_whence	SEEK_SET.
l_start	Start of the blocking lock.
l_len	Length of the blocking lock.
l_pid	Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall be determined before the *fcntl*( ) function blocks. If the file size or file descriptor seek offset change while *fcntl*( ) is blocked, this shall not affect the range of bytes locked.

If *l_len* is positive, the area affected shall start at *l_start* and end at $l\_start+l\_len-1$. If *l_len* is negative, the area affected shall start at $l\_start+l\_len$ and end at $l\_start-1$. Locks may start and extend beyond the current end of a file, but shall not extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting *l_len* to 0. If such a lock also has *l_start* set to 0 and *l_whence* is set to SEEK_SET, the whole file shall be locked.

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, *fcntl*( ) shall fail with an [EDEADLK] error.

An unlock (F_UNLCK) request in which *l_len* is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type **off_t**, when the process has an existing lock in which *l_len* is 0 and which includes the last byte of the requested segment, shall be treated as a request to unlock from the start of the requested segment with an *l_len* equal to 0. Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.

SHM When the file descriptor *fildes* refers to a shared memory object, the behavior of *fcntl*( ) shall be the same as for a regular file except the effect of the following values for the argument *cmd* shall be unspecified: F_SETFL, F_GETLK, F_SETLK, and F_SETLKW.

TYM If *fildes* refers to a typed memory object, the result of the *fcntl*( ) function is unspecified.

RETURN VALUE

Upon successful completion, the value returned shall depend on *cmd* as follows:

F_DUPFD	A new file descriptor.
F_GETFD	Value of flags defined in **<fcntl.h>**. The return value shall not be negative.
F_SETFD	Value other than −1.
F_GETFL	Value of file status flags and access modes. The return value is not negative.
F_SETFL	Value other than −1.
F_GETLK	Value other than −1.
F_SETLK	Value other than −1.
F_SETLKW	Value other than −1.
F_GETOWN	Value of the socket owner process or process group; this will not be −1.

F_SETOWN	Value other than –1.

Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fcntl*( ) function shall fail if:

[EACCES] or [EAGAIN]
The *cmd* argument is F_SETLK; the type of lock (*l_type*) is a shared (F_RDLCK) or exclusive (F_WRLCK) lock and the segment of a file to be locked is already exclusive-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

[EBADF]	The *fildes* argument is not a valid open file descriptor, or the argument *cmd* is F_SETLK or F_SETLKW, the type of lock, *l_type*, is a shared lock (F_RDLCK), and *fildes* is not a valid file descriptor open for reading, or the type of lock *l_type*, is an exclusive lock (F_WRLCK), and *fildes* is not a valid file descriptor open for writing.
[EINTR]	The *cmd* argument is F_SETLKW and the function was interrupted by a signal.
[EINVAL]	The *cmd* argument is invalid, or the *cmd* argument is F_DUPFD and *arg* is negative or greater than or equal to {OPEN_MAX}, or the *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the data pointed to by *arg* is not valid, or *fildes* refers to a file that does not support locking.
[EMFILE]	The argument *cmd* is F_DUPFD and {OPEN_MAX} file descriptors are currently open in the calling process, or no file descriptors greater than or equal to *arg* are available.
[ENOLCK]	The argument *cmd* is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.
[EOVERFLOW]	One of the values to be returned cannot be represented correctly.
[EOVERFLOW]	The *cmd* argument is F_GETLK, F_SETLK, or F_SETLKW and the smallest or, if *l_len* is non-zero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type **off_t**.

The *fcntl*( ) function may fail if:

[EDEADLK]	The *cmd* argument is F_SETLKW, the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The ellipsis in the SYNOPSIS is the syntax specified by the ISO C standard for a variable number of arguments. It is used because System V uses pointers for the implementation of file locking functions.

The *arg* values to F_GETFD, F_SETFD, F_GETFL, and F_SETFL all represent flag values to allow for future growth. Applications using these functions should do a read-modify-write operation on them, rather than assuming that only the values defined by this volume of IEEE Std 1003.1-200x are valid. It is a common error to forget this, particularly in the case of F_SETFD.

This volume of IEEE Std 1003.1-200x permits concurrent read and write access to file data using the *fcntl*( ) function; this is a change from the 1984 /usr/group standard and early proposals. Without concurrency controls, this feature may not be fully utilized without occasional loss of data.

Data losses occur in several ways. One case occurs when several processes try to update the same record, without sequencing controls; several updates may occur in parallel and the last writer ‘‘wins’’. Another case is a bit-tree or other internal list-based database that is undergoing reorganization. Without exclusive use to the tree segment by the updating process, other reading processes chance getting lost in the database when the index blocks are split, condensed, inserted, or deleted. While *fcntl*( ) is useful for many applications, it is not intended to be overly general and does not handle the bit-tree example well.

This facility is only required for regular files because it is not appropriate for many devices such as terminals and network connections.

Since *fcntl*( ) works with ‘‘any file descriptor associated with that file, however it is obtained’’, the file descriptor may have been inherited through a *fork*( ) or *exec* operation and thus may affect a file that another process also has open.

The use of the open file description to identify what to lock requires extra calls and presents problems if several processes are sharing an open file description, but there are too many implementations of the existing mechanism for this volume of IEEE Std 1003.1-200x to use different specifications.

Another consequence of this model is that closing any file descriptor for a given file (whether or not it is the same open file description that created the lock) causes the locks on that file to be relinquished for that process. Equivalently, any close for any file/process pair relinquishes the locks owned on that file for that process. But note that while an open file description may be shared through *fork*( ), locks are not inherited through *fork*( ). Yet locks may be inherited through one of the *exec* functions.

The identification of a machine in a network environment is outside of the scope of this volume of IEEE Std 1003.1-200x. Thus, an *l_sysid* member, such as found in System V, is not included in the locking structure.

Changing of lock types can result in a previously locked region being split into smaller regions.

Mandatory locking was a major feature of the 1984 /usr/group standard.

For advisory file record locking to be effective, all processes that have access to a file must cooperate and use the advisory mechanism before doing I/O on the file. Enforcement-mode record locking is important when it cannot be assumed that all processes are cooperating. For example, if one user uses an editor to update a file at the same time that a second user executes another process that updates the same file and if only one of the two processes is using advisory locking, the processes are not cooperating. Enforcement-mode record locking would protect against accidental collisions.

Secondly, advisory record locking requires a process using locking to bracket each I/O operation with lock (or test) and unlock operations. With enforcement-mode file and record locking, a process can lock the file once and unlock when all I/O operations have been completed. Enforcement-mode record locking provides a base that can be enhanced; for example, with sharable locks. That is, the mechanism could be enhanced to allow a process to lock a file so other processes could read it, but none of them could write it.

Mandatory locks were omitted for several reasons:

1. Mandatory lock setting was done by multiplexing the set-group-ID bit in most implementations; this was confusing, at best.
2. The relationship to file truncation as supported in 4.2 BSD was not well specified.
3. Any publicly readable file could be locked by anyone. Many historical implementations keep the password database in a publicly readable file. A malicious user could thus prohibit logins. Another possibility would be to hold open a long-distance telephone line.
4. Some demand-paged historical implementations offer memory mapped files, and enforcement cannot be done on that type of file.

Since sleeping on a region is interrupted with any signal, *alarm*( ) may be used to provide a timeout facility in applications requiring it. This is useful in deadlock detection. Since implementation of full deadlock detection is not always feasible, the [EDEADLK] error was made optional.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *exec*, *open*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<signal.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION, sentences describing behavior when *l_len* is negative are now mandated, and the description of unlock (F_UNLOCK) when *l_len* is non-negative is mandated.
- In the ERRORS section, the [EINVAL] error condition has the case mandated when the *cmd* is invalid, and two [EOVERFLOW] error conditions are added.

The F_GETOWN and F_SETOWN values are added for sockets.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that the extent of the bytes locked is determined prior to the blocking action.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *fcntl*( ) results are unspecified for typed memory objects.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

fcvt — convert a floating-point number to a string (**LEGACY**)

SYNOPSIS

XSI

```
#include <stdlib.h>

char *fcvt(double value, int ndigit, int *restrict decpt,
    int *restrict sign);
```

DESCRIPTION

Refer to *ecvt*( ).

NAME

fdatasync — synchronize the data of a file (**REALTIME**)

SYNOPSIS

SIO

```
#include <unistd.h>

int fdatasync(int fildes);
```

DESCRIPTION

The *fdatasync*( ) function shall force all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state.

The functionality shall be equivalent to *fsync*( ) with the symbol _POSIX_SYNCHRONIZED_IO defined, with the exception that all I/O operations shall be completed as defined for synchronized I/O data integrity completion. |

RETURN VALUE

If successful, the *fdatasync*( ) function shall return the value 0; otherwise, the function shall return the value −1 and set *errno* to indicate the error. If the *fdatasync*( ) function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRORS

The *fdatasync*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor open for writing.
[EINVAL]	This implementation does not support synchronized I/O for this file.

In the event that any of the queued I/O operations fail, *fdatasync*( ) shall return the error conditions defined for *read*( ) and *write*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_fsync( ), *fcntl*( ), *fsync*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Synchronized Input and Output option.

The *fdatasync*( ) function is marked as part of the Synchronized Input and Output option.

NAME

fdetach — detach a name from a STREAMS-based file descriptor (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>

int fdetach(const char *path);
```

DESCRIPTION

The *fdetach*( ) function shall detach a STREAMS-based file from the file to which it was attached by a previous call to *fattach*( ). The *path* argument points to the pathname of the attached STREAMS file. The process shall have appropriate privileges or be the owner of the file. A successful call to *fdetach*( ) shall cause all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* shall operate on the underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file referenced by *path* shall still refer to the STREAMS file after the *fdetach*( ) has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to *fdetach*( ) shall have be equivalent to performing the last *close*( ) on the attached file.

RETURN VALUE

Upon successful completion, *fdetach*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *fdetach*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix.
[EINVAL]	The *path* argument names a file that is not currently attached.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The size of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The effective user ID is not the owner of *path* and the process does not have appropriate privileges.

The *fdetach*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Detaching a File

The following example detaches the STREAMS-based file **/tmp/named-STREAM** from the file to which it was attached by a previous, successful call to *fattach*( ). Subsequent calls to open this file refer to the underlying file, not to the STREAMS file.

```
#include <stropts.h>
...
    char *filename = "/tmp/named-STREAM";
    int ret;

    ret = fdetach(filename);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fattach( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

fdim, fdimf, fdiml — compute positive difference between two floating-point numbers

SYNOPSIS

```
#include <math.h>

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall determine the positive difference between their arguments. If *x* is greater than *y*, $x-y$ is returned. If *x* is less than or equal to *y*, +0 is returned.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the positive difference value.

If $x-y$ is positive and overflows, a range error shall occur and *fdim*( ), *fdimf*( ), and *fdiml*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

XSI If $x-y$ is positive and underflows, a range error may occur, and either $(x-y)$ (if representable), or 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* or *y* is NaN, a NaN shall be returned.

ERRORS

The *fdim*( ) function shall fail if:

Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

The *fdim*( ) function may fail if:

Range Error The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On implementations supporting IEEE Std 754-1985, $x-y$ cannot underflow, and hence the 0.0 return value is shaded as an extension for implementations supporting the XSI extension rather than an MX extension.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *fmax*( ), *fmin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

fdopen — associate a stream with a file descriptor

SYNOPSIS

CX

```
#include <stdio.h>

FILE *fdopen(int fildes, const char *mode);
```

DESCRIPTION

The *fdopen*( ) function shall associate a stream with a file descriptor.

The *mode* argument is a character string having one of the following values:

r or *rb*	Open a file for reading.
w or *wb*	Open a file for writing.
a or *ab*	Open a file for writing at end of file.
r+ or *rb*+ or *r*+*b*	Open a file for update (reading and writing).
w+ or *wb*+ or *w*+*b*	Open a file for update (reading and writing).
a+ or *ab*+ or *a*+*b*	Open a file for update (reading and writing) at end of file.

The meaning of these flags is exactly as specified in *fopen*( ), except that modes beginning with *w* shall not cause truncation of the file.

Additional values for the *mode* argument may be supported by an implementation.

The application shall ensure that the mode of the stream as expressed by the *mode* argument is allowed by the file access mode of the open file description to which *fildes* refers. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

The error and end-of-file indicators for the stream shall be cleared. The *fdopen*( ) function may cause the *st_atime* field of the underlying file to be marked for update.

SHM If *fildes* refers to a shared memory object, the result of the *fdopen*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *fdopen*( ) function is unspecified.

The *fdopen*( ) function shall preserve the offset maximum previously set for the open file description corresponding to *fildes*.

RETURN VALUE

Upon successful completion, *fdopen*( ) shall return a pointer to a stream; otherwise, a null pointer shall be returned and *errno* set to indicate the error.

ERRORS

The *fdopen*( ) function may fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *mode* argument is not a valid mode.
[EMFILE]	{FOPEN_MAX} streams are currently open in the calling process.
[EMFILE]	{STREAM_MAX} streams are currently open in the calling process.
[ENOMEM]	Insufficient space to allocate a buffer.

EXAMPLES

None.

APPLICATION USAGE

File descriptors are obtained from calls like *open*( ), *dup*( ), *creat*( ), or *pipe*( ), which open files but do not return streams.

RATIONALE

The file descriptor may have been obtained from *open*( ), *creat*( ), *pipe*( ), *dup*( ), or *fcntl*( ); inherited through *fork*( ) or *exec*; or perhaps obtained by implementation-defined means, such as the 4.3 BSD *socket*( ) call.

The meanings of the *mode* arguments of *fdopen*( ) and *fopen*( ) differ. With *fdopen*( ), open for write (*w* or *w*+) does not truncate, and append (*a* or *a*+) cannot create for writing. The *mode* argument formats that include a *b* are allowed for consistency with the ISO C standard function *fopen*( ). The *b* has no effect on the resulting stream. Although not explicitly required by this volume of IEEE Std 1003.1-200x, a good implementation of append (*a*) mode would cause the O_APPEND flag to be set.

FUTURE DIRECTIONS

None.

SEE ALSO

fclose( ), *fopen*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, Section 2.5.1 (on page 485)

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the use and setting of the *mode* argument are changed to include binary streams.
- In the DESCRIPTION, text is added for large file support to indicate setting of the offset maximum in the open file description.
- All errors identified in the ERRORS section are added.
- In the DESCRIPTION, text is added that the *fdopen*( ) function may cause *st_atime* to be updated.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Clarification is added that it is the responsibility of the application to ensure that the mode is compatible with the open file descriptor.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *fdopen*( ) results are unspecified for typed memory objects.

NAME

feclearexcept — clear floating-point exception

SYNOPSIS

```
#include <fenv.h>

int feclearexcept(int excepts);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *feclearexcept*( ) function shall attempt to clear the supported floating-point exceptions represented by *excepts*. |

RETURN VALUE

If the argument is zero or if all the specified exceptions were successfully cleared, *feclearexcept*( ) shall return zero. Otherwise, it shall return a non-zero value. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fegetexceptflag( ), *feraiseexcept*( ), *fesetexceptflag*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard. |

ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

fegetenv, fesetenv — get and set current floating-point environment

SYNOPSIS

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fegetenv*( ) function shall attempt to store the current floating-point environment in the object pointed to by *envp*. |

The *fesetenv*( ) function shall attempt to establish the floating-point environment represented by the object pointed to by *envp*. The argument *envp* shall point to an object set by a call to *fegetenv*( ) or *feholdexcept*( ), or equal a floating-point environment macro. The *fesetenv*( ) function does not raise floating-point exceptions, but only installs the state of the floating-point status flags represented through its argument. |

RETURN VALUE

If the representation was successfully stored, *fegetenv*( ) shall return zero. Otherwise, it shall return a non-zero value. If the environment was successfully established, *fesetenv*( ) shall return zero. Otherwise, it shall return a non-zero value. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feholdexcept( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard. |

ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

fegetexceptflag, fesetexceptflag — get and set floating-point status flags

SYNOPSIS

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fegetexceptflag*( ) function shall attempt to store an implementation-defined representation of the states of the floating-point status flags indicated by the argument *excepts* in the object pointed to by the argument *flagp*.

The *fesetexceptflag*( ) function shall attempt to set the floating-point status flags indicated by the argument excepts to the states stored in the object pointed to by *flagp*. The value pointed to by *flagp* shall have been set by a previous call to *fegetexceptflag*( ) whose second argument represented at least those floating-point exceptions represented by the argument *excepts*. This function does not raise floating-point exceptions, but only sets the state of the flags.

RETURN VALUE

If the representation was successfully stored, *fegetexceptflag*( ) shall return zero. Otherwise, it shall return a non-zero value. If the *excepts* argument is zero or if all the specified execptions were successfully set, *fesetexceptflag*( ) shall return zero. Otherwise, it shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *feraiseexcept*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

NAME

fegetround, fesetround — get and set current rounding direction

SYNOPSIS

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fegetround*( ) function shall get the current rounding direction.

The *fesetround*( ) function shall establish the rounding direction represented by its argument *round*. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

RETURN VALUE

The *fegetround*( ) function shall return the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The *fesetround*( ) function shall return a zero value if and only if the requested rounding direction was established. |

ERRORS

No errors are defined.

EXAMPLES

The following example saves, sets, and restores the rounding direction, reporting an error and aborting if setting the rounding direction fails:

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

NAME

feholdexcept — save current floating-point environment

SYNOPSIS

```
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *feholdexcept*( ) function shall save the current floating-point environment in the object pointed to by *envp*, clear the floating-point status flags, and then install a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.

RETURN VALUE

The *feholdexcept*( ) function shall return zero if and only if non-stop floating-point exception handling was successfully installed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *feholdexcept*( ) function should be effective on typical IEC 60559:1989 standard implementations which have the default non-stop mode and at least one other mode for trap handling or aborting. If the implementation provides only the non-stop mode, then installing the non-stop mode is trivial.

FUTURE DIRECTIONS

None.

SEE ALSO

fegetenv( ), *fesetenv*( ), *feupdateenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

feof — test end-of-file indicator on a stream

SYNOPSIS

```
#include <stdio.h>

int feof(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *feof*( ) function shall test the end-of-file indicator for the stream pointed to by *stream*.

RETURN VALUE

The *feof*( ) function shall return non-zero if and only if the end-of-file indicator is set for *stream*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clearerr( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

feraiseexcept — raise floating-point exception

SYNOPSIS

```
#include <fenv.h>

int feraiseexcept(int excepts);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *feraiseexcept*( ) function shall attempt to raise the supported floating-point exceptions represented by the argument *excepts*. The order in which these floating-point exceptions are raised is unspecified. Whether the *feraiseexcept*( ) function additionally raises the inexact floating-point exception whenever it raises the overflow or underflow floating-point exception is implementation-defined. |

RETURN VALUE

If the argument is zero or if all the specified exceptions were successfully raised, *feraiseexcept*( ) shall return zero. Otherwise, it shall return a non-zero value. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The effect is intended to be similar to that of floating-point exceptions raised by arithmetic operations. Hence, enabled traps for floating-point exceptions raised by this function are taken.

RATIONALE

Raising overflow or underflow is allowed to also raise inexact because on some architectures the only practical way to raise an exception is to execute an instruction that has the exception as a side effect. The function is not restricted to accept only valid coincident expressions for atomic operations, so the function can be used to raise exceptions accrued over several operations.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fegetexceptflag*( ), *fesetexceptflag*( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard. |

ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

ferror — test error indicator on a stream

SYNOPSIS

```
#include <stdio.h>

int ferror(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ferror*( ) function shall test the error indicator for the stream pointed to by *stream*.

RETURN VALUE

The *ferror*( ) function shall return non-zero if and only if the error indicator is set for *stream*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clearerr( ), *feof*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

fesetenv — set current floating-point environment

SYNOPSIS

```
#include <fenv.h>

int fesetenv(const fenv_t *envp);
```

DESCRIPTION

Refer to *fegetenv*( ).

NAME

fesetexceptflag — set floating-point status flags

SYNOPSIS

```
#include <fenv.h>

int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

DESCRIPTION

Refer to *fegetexceptflag*( ).

NAME

fesetround — set current rounding direction

SYNOPSIS

```
#include <fenv.h>

int fesetround(int round);
```

DESCRIPTION

Refer to *fegetround*( ).

NAME

fetestexcept — test floating-point exception flags

SYNOPSIS

```
#include <fenv.h>

int fetestexcept(int excepts);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fetestexcept*( ) function shall determine which of a specified subset of the floating-point exception flags are currently set. The *excepts* argument specifies the floating-point status flags to be queried.

RETURN VALUE

The *fetestexcept*( ) function shall return the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in *excepts*.

ERRORS

No errors are defined.

EXAMPLES

The following example calls function *f*( ) if an invalid exception is set, and then function *g*( ) if an overflow exception is set:

```
#include <fenv.h>
/* ... */
{
    #pragma STDC FENV_ACCESS ON
    int set_excepts;
    feclearexcept(FE_INVALID | FE_OVERFLOW);
    // maybe raise exceptions
    set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
    if (set_excepts & FE_INVALID) f();
    if (set_excepts & FE_OVERFLOW) g();
    /* ... */
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fegetexceptflag*( ), *feraiseexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

feupdateenv — update floating-point environment

SYNOPSIS

```
#include <fenv.h>

int feupdateenv(const fenv_t *envp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *feupdateenv*( ) function shall attempt to save the currently raised floating-point exceptions in its automatic storage, attempt to install the floating-point environment represented by the object pointed to by *envp*, and then attempt to raise the saved floating-point exceptions. The argument *envp* shall point to an object set by a call to *feholdexcept*( ) or *fegetenv*( ), or equal a floating-point environment macro.

RETURN VALUE

The *feupdateenv*( ) function shall return a zero value if and only if all the required actions were successfully carried out.

ERRORS

No errors are defined.

EXAMPLES

The following example shows sample code to hide spurious underflow floating-point exceptions:

```
#include <fenv.h>
double f(double x)
{
    #pragma STDC FENV_ACCESS ON
    double result;
    fenv_t save_env;
    feholdexcept(&save_env);
    // compute result
    if (/* test spurious underflow */)
    feclearexcept(FE_UNDERFLOW);
    feupdateenv(&save_env);
    return result;
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fegetenv( ), *feholdexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fenv.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard. |

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

fflush — flush a stream

SYNOPSIS

```
#include <stdio.h>

int fflush(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX If *stream* points to an output stream or an update stream in which the most recent operation was not input, *fflush*( ) shall cause any unwritten data for that stream to be written to the file, and the *st_ctime* and *st_mtime* fields of the underlying file shall be marked for update.

If *stream* is a null pointer, *fflush*( ) shall perform this flushing action on all streams for which the behavior is defined above.

RETURN VALUE

CX Upon successful completion, *fflush*( ) shall return 0; otherwise, it shall set the error indicator for the stream, return EOF, and set *errno* to indicate the error.

ERRORS

The *fflush*( ) function shall fail if:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
CX	[EBADF]	The file descriptor underlying *stream* is not valid.
CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.
CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
CX	[EINTR]	The *fflush*( ) function was interrupted by a signal.
CX	[EIO]	The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
CX	[ENOSPC]	There was no free space remaining on the device containing the file.
CX	[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The *fflush*( ) function may fail if:

CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

Sending Prompts to Standard Output

The following example uses *printf*( ) calls to print a series of prompts for information the user must enter from standard input. The *fflush*( ) calls force the output to standard output. The *fflush*( ) function is used because standard output is usually buffered and the prompt may not immediately be printed on the output or terminal. The *gets*( ) calls read strings from standard input and place the results in variables, for use later in the program

```
#include <stdio.h>
...
char user[100];
char oldpasswd[100];
char newpasswd[100];
...
printf("User name: ");
fflush(stdout);
gets(user);

printf("Old password: ");
fflush(stdout);
gets(oldpasswd);

printf("New password: ");
fflush(stdout);
gets(newpasswd);
...
```

APPLICATION USAGE

None.

RATIONALE

Data buffered by the system may make determining the validity of the position of the current file descriptor impractical. Thus, enforcing the repositioning of the file descriptor after *fflush*( ) on streams open for *read*( ) is not mandated by IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

getrlimit( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] error is added as part of the large file support extensions.
- The [ENXIO] optional error condition is added.

The RETURN VALUE section is updated to note that the error indicator shall be set for the stream. This is for alignment with the ISO/IEC 9899:1999 standard.

NAME

ffs — find first set bit

SYNOPSIS

XSI

```
#include <strings.h>

int ffs(int i);
```

DESCRIPTION

The *ffs*( ) function shall find the first bit set (beginning with the least significant bit) in *i*, and return the index of that bit. Bits are numbered starting at one (the least significant bit).

RETURN VALUE

The *ffs*( ) function shall return the index of the first bit set. If *i* is 0, then *ffs*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

fgetc — get a byte from a stream

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If the end-of-file indicator for the input stream pointed to by *stream* is not set and a next byte is present, the *fgetc*( ) function shall obtain the next byte as an **unsigned char** converted to an **int**, from the input stream pointed to by *stream*, and advance the associated file position indicator for the stream (if defined). Since *fgetc*( ) operates on bytes, reading a character consisting of multiple bytes (or ‘‘a multi-byte character’’) may require multiple calls to *fgetc*( ).

CX The *fgetc*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, *fgetc*( ) shall return the next byte from the input stream pointed to by *stream*. If the end-of-file indicator for the stream is set, or if the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetc*( ) shall return EOF. If a read error occurs, the error indicator for the stream shall be set, *fgetc*( ) shall return EOF, CX and shall set *errno* to indicate the error.

ERRORS

The *fgetc*( ) function shall fail if data needs to be read and:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the *fgetc*( ) operation.
CX	[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for reading.
CX	[EINTR]	The read operation was terminated due to the receipt of a signal, and no data was transferred.
CX	[EIO]	A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned. This error may also be generated for implementation-defined reasons.
CX	[EOVERFLOW]	The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The *fgetc*( ) function may fail if:

CX	[ENOMEM]	Insufficient storage space is available.
CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

If the integer value returned by *fgetc*( ) is stored into a variable of type **char** and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type **char** on widening to integer is implementation-defined.

The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an end-of-file condition.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof( ), *ferror*( ), *fopen*( ), *getchar*( ), *getc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The DESCRIPTION is updated to clarify the behavior when the end-of-file indicator for the input stream is not set.
- The RETURN VALUE section is updated to note that the error indicator shall be set for the stream.

NAME

fgetpos — get current file position information

SYNOPSIS

```
#include <stdio.h>

int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fgetpos*( ) function shall store the current values of the parse state (if any) and file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains unspecified information usable by *fsetpos*( ) for repositioning the stream to its position at the time of the call to *fgetpos*( ).

RETURN VALUE

Upon successful completion, *fgetpos*( ) shall return 0; otherwise, it shall return a non-zero value and set *errno* to indicate the error.

ERRORS

The *fgetpos*( ) function shall fail if:

CX [EOVERFLOW]	The current value of the file position cannot be represented correctly in an object of type **fpos_t**.

The *fgetpos*( ) function may fail if:

CX [EBADF]	The file descriptor underlying *stream* is not valid.
CX [ESPIPE]	The file descriptor underlying *stream* is associated with a pipe, FIFO, or socket.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *ftell*( ), *rewind*( ), *ungetc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] mandatory error condition is added.
- The [EBADF] and [ESPIPE] optional error conditions are added.

An additional [ESPIPE] error condition is added for sockets.

The prototype for *fgetpos*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fgets — get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *fgets(char *restrict s, int n, FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fgets*( ) function shall read bytes from *stream* into the array pointed to by *s*, until *n*−1 bytes are read, or a <newline> is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null byte.

CX The *fgets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, *fgets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgets*( ) shall return a null pointer. If a read error occurs, CX the error indicator for the stream shall be set, *fgets*( ) shall return a null pointer, and shall set *errno* to indicate the error.

ERRORS

Refer to *fgetc*( ).

EXAMPLES

Reading Input

The following example uses *fgets*( ) to read each line of input. {LINE_MAX}, which defines the maximum size of the input line, is defined in the **<limits.h>** header.

```
#include <stdio.h>
...
char line[LINE_MAX];
...
while (fgets(line, LINE_MAX, fp) != NULL) {
...
}
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *fread*( ), *gets*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The prototype for *fgets*( ) is changed for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fgetwc — get a wide-character code from a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t fgetwc(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fgetwc*( ) function shall obtain the next character (if present) from the input stream pointed to by *stream*, convert that to the corresponding wide-character code, and advance the associated file position indicator for the stream (if defined).

If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

CX The *fgetwc*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, the *fgetwc*( ) function shall return the wide-character code of the character read from the input stream pointed to by *stream* converted to a type **wint_t**. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetwc*( ) shall return WEOF. If a read error occurs, the error indicator for the stream shall be set, *fgetwc*( ) shall return WEOF, CX and shall set *errno* to indicate the error. If an encoding error occurs, the error indicator for the stream shall be set, *fgetwc*( ) shall return WEOF, and shall set *errno* to indicate the error.

ERRORS

The *fgetwc*( ) function shall fail if data needs to be read and:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the *fgetwc*( ) operation.
CX	[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for reading.
	[EILSEQ]	The data obtained from the input stream does not form a valid character.
CX	[EINTR]	The read operation was terminated due to the receipt of a signal, and no data was transferred.
CX	[EIO]	A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group is orphaned. This error may also be generated for implementation-defined reasons.
CX	[EOVERFLOW]	The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The *fgetwc*( ) function may fail if:

CX	[ENOMEM]	Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an end-of-file condition.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof( ), *ferror*( ), *fopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EOVERFLOW] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

NAME

fgetws — get a wide-character string from a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wchar_t *fgetws(wchar_t *restrict ws, int n,
    FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fgetws*( ) function shall read characters from the *stream*, convert these to the corresponding wide-character codes, place them in the **wchar_t** array pointed to by *ws*, until *n*−1 characters are read, or a <newline> is read, converted, and transferred to *ws*, or an end-of-file condition is encountered. The wide-character string, *ws*, shall then be terminated with a null wide-character code.

If an error occurs, the resulting value of the file position indicator for the stream is unspecified.

CX The *fgetws*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, *fgetws*( ) shall return *ws*. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *fgetws*( ) shall return a null pointer. If a read error occurs, the error indicator for the stream shall be set, *fgetws*( ) shall return a null pointer, CX and shall set *errno* to indicate the error.

ERRORS

Refer to *fgetwc*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *fread*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

Issue 6

Extensions beyond the ISO C standard are now marked.

The prototype for *fgetws*( ) is changed for alignment with the ISO/IEC 9899:1999 standard.

NAME

fileno — map a stream pointer to a file descriptor

SYNOPSIS

CX
```
#include <stdio.h>

int fileno(FILE *stream);
```

DESCRIPTION

The *fileno*( ) function shall return the integer file descriptor associated with the stream pointed to by *stream*.

RETURN VALUE

Upon successful completion, *fileno*( ) shall return the integer value of the file descriptor associated with *stream*. Otherwise, the value −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fileno*( ) function may fail if:

[EBADF]	The *stream* argument is not a valid stream.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Without some specification of which file descriptors are associated with these streams, it is impossible for an application to set up the streams for another application it starts with *fork*( ) and *exec*. In particular, it would not be possible to write a portable version of the *sh* command interpreter (although there may be other constraints that would prevent that portability).

FUTURE DIRECTIONS

None.

SEE ALSO

fdopen( ), *fopen*( ), *stdin*, the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, Section 2.5.1 (on page 485)

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EBADF] optional error condition is added.

NAME

flockfile, ftrylockfile, funlockfile — stdio locking functions

SYNOPSIS

TSF

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

DESCRIPTION

These functions shall provide for explicit application-level locking of stdio (**FILE ***) objects. These functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

The *flockfile*( ) function shall acquire for a thread ownership of a (**FILE ***) object.

The *ftrylockfile*( ) function shall acquire for a thread ownership of a (**FILE ***) object if the object is available; *ftrylockfile*( ) is a non-blocking version of *flockfile*( ).

The *funlockfile*( ) function shall relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the *funlockfile*( ) function.

The functions shall behave as if there is a lock count associated with each (**FILE ***) object. This count is implicitly initialized to zero when the (**FILE ***) object is created. The (**FILE ***) object is unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE ***) object. When the *flockfile*( ) function is called, if the count is zero or if the count is positive and the caller owns the (**FILE ***) object, the count shall be incremented. Otherwise, the calling thread shall be suspended, waiting for the count to return to zero. Each call to *funlockfile*( ) shall decrement the count. This allows matching calls to *flockfile*( ) (or successful calls to *ftrylockfile*( )) and *funlockfile*( ) to be nested.

All functions that reference (**FILE ***) objects shall behave as if they use *flockfile*( ) and *funlockfile*( ) internally to obtain ownership of these (**FILE ***) objects.

RETURN VALUE

None for *flockfile*( ) and *funlockfile*( ). The *ftrylockfile*( ) function shall return zero for success and non-zero to indicate that the lock cannot be acquired.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

The *flockfile*( ) and *funlockfile*( ) functions provide an orthogonal mutual exclusion lock for each **FILE**. The *ftrylockfile*( ) function provides a non-blocking attempt to acquire a file lock, analogous to *pthread_mutex_trylock*( ).

These locks behave as if they are the same as those used internally by *stdio* for thread-safety. This both provides thread-safety of these functions without requiring a second level of internal locking and allows functions in *stdio* to be implemented in terms of other *stdio* functions.

Application writers and implementors should be aware that there are potential deadlock problems on **FILE** objects. For example, the line-buffered flushing semantics of *stdio* (requested via {_IOLBF}) require that certain input operations sometimes cause the buffered contents of implementation-defined line-buffered output streams to be flushed. If two threads each hold the lock on the other's **FILE**, deadlock ensues. This type of deadlock can be avoided by acquiring **FILE** locks in a consistent order. In particular, the line-buffered output stream deadlock can typically be avoided by acquiring locks on input streams before locks on output streams if a thread would be acquiring both.

In summary, threads sharing *stdio* streams with other threads can use *flockfile*( ) and *funlockfile*( ) to cause sequences of I/O performed by a single thread to be kept bundled. The only case where the use of *flockfile*( ) and *funlockfile*( ) is required is to provide a scope protecting uses of the *_unlocked*( ) functions/macros. This moves the cost/performance tradeoff to the optimal point.

FUTURE DIRECTIONS

None.

SEE ALSO

getc_unlocked( ), *putc_unlocked*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Thread-Safe Functions option.

NAME

floor, floorf, floorl — floor function

SYNOPSIS

```
#include <math.h>

double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the largest integral value not greater than *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the largest integral value not greater than *x*, expressed as a **double**, **float**, or **long double**, as appropriate for the return type of the function.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0 or ±Inf, *x* shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *floor*( ), *floorf*( ), and *floorl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

XSI Range Error The result would cause an overflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

The integral value returned by these functions might not be expressible as an **int** or **long**. The return value should be tested before assigning it to an integer type to avoid the undefined results of an integer overflow.

The *floor*( ) function can only overflow when the floating-point representation has DBL_MANT_DIG > DBL_MAX_EXP.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ceil( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *floorf*( ) and *floorl*( ) functions are added for alignment with the ISO/IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

NAME

fma, fmaf, fmal — floating-point multiply-add

SYNOPSIS

```
#include <math.h>

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute $(x * y) + z$, rounded as one ternary operation: they shall compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return $(x * y) + z$, rounded as one ternary operation.

MX If *x* or *y* are NaN, a NaN shall be returned.

If *x* multiplied by *y* is an exact infinity and *z* is also an infinity but with the opposite sign, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If one of *x* and *y* is infinite, the other is zero, and *z* is not a NaN, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If one of *x* and *y* is infinite, the other is zero, and *z* is a NaN, a NaN shall be returned and a domain error may occur.

If $x*y$ is not 0*Inf nor Inf*0 and *z* is a NaN, a NaN shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error	The value of $x*y+z$ is invalid, or the value $x*y$ is invalid and *z* is not a NaN.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
MX Range Error	The result overflows.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

MX	Domain Error	The value *x***y* is invalid and *z* is a NaN. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
MX	Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

In many cases, clever use of floating (*fused*) multiply-add leads to much improved code; but its unexpected use by the compiler can undermine carefully written code. The FP_CONTRACT macro can be used to disallow use of floating multiply-add; and the *fma*( ) function guarantees its use where desired. Many current machines provide hardware floating multiply-add instructions; software implementation can be used for others.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

fmax, fmaxf, fmaxl — determine maximum numeric value of two floating-point numbers

SYNOPSIS

```
#include <math.h>

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall determine the maximum numeric value of their arguments. NaN arguments shall be treated as missing data: if one argument is a NaN and the other numeric, then these functions shall choose the numeric value.

RETURN VALUE

Upon successful completion, these functions shall return the maximum numeric value of their arguments.

If just one argument is a NaN, the other argument shall be returned.

MX If *x* and *y* are NaN, a NaN shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fdim( ), *fmin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

fmin, fminf, fminl — determine minimum numeric value of two floating-point numbers

SYNOPSIS

```
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall determine the minimum numeric value of their arguments. NaN arguments shall be treated as missing data: if one argument is a NaN and the other numeric, then these functions shall choose the numeric value.

RETURN VALUE

Upon successful completion, these functions shall return the minimum numeric value of their arguments.

If just one argument is a NaN, the other argument shall be returned.

MX If *x* and *y* are NaN, a NaN shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fdim( ), *fmax*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from ISO/IEC 9899:1999 standard.

NAME

fmod, fmodf, fmodl — floating-point remainder value function

SYNOPSIS

```
#include <math.h>

double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall return the floating-point remainder of the division of *x* by *y*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

These functions shall return the value $x-i*y$, for some integer *i* such that, if *y* is non-zero, the result has the same sign as *x* and magnitude less than the magnitude of *y*.

If the correct value would cause underflow, and is not representable, a range error may occur, MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* or *y* is NaN, a NaN shall be returned.

If *y* is zero, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If *x* is infinite, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If *x* is ±0 and *y* is not zero, ±0 shall be returned.

If *x* is not infinite and *y* is ±Inf, *x* shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error	The *x* argument is infinite or *y* is zero.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

These functions may fail if:

Range Error	The result underflows.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The behavior for when the *y* argument is zero is now defined.

The *fmodf*( ) and *fmodl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

fmtmsg — display a message in the specified format on standard error and/or a system console

SYNOPSIS

XSI

```
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity,
    const char *text, const char *action, const char *tag);
```

DESCRIPTION

The *fmtmsg*( ) function shall display messages in a specified format instead of the traditional *printf*( ) function.

Based on a message's classification component, *fmtmsg*( ) shall write a formatted message either to standard error, to the console, or to both.

A formatted message consists of up to five components as defined below. The component *classification* is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

classification Contains the sum of identifying values constructed from the constants defined below. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and the system console).

Major Classifications
Identifies the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).

Message Source Subclassifications
Identifies the type of software in which the problem is detected. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).

Display Subclassifications
Indicates where the message is to be displayed. Identifiers are: MM_PRINT to display the message on the standard error stream, MM_CONSOLE to display the message on the system console. One or both identifiers may be used.

Status Subclassifications
Indicates whether the application can recover from the condition. Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-recoverable).

An additional identifier, MM_NULLMC, indicates that no classification component is supplied for the message.

label Identifies the source of the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second is up to 14 bytes.

severity Indicates the seriousness of the condition. Identifiers for the levels of *severity* are:

MM_HALT	Indicates that the application has encountered a severe fault and is halting. Produces the string `"HALT"`.
MM_ERROR	Indicates that the application has detected a fault. Produces the string `"ERROR"`.
MM_WARNING	Indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string `"WARNING"`.
MM_INFO	Provides information about a condition that is not in error. Produces the string `"INFO"`.
MM_NOSEV	Indicates that no severity level is supplied for the message.

text	Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.
action	Describes the first step to be taken in the error-recovery process. The *fmtmsg*( ) function precedes the action string with the prefix: `"TO FIX:"`. The *action* string is not limited to a specific size.
tag	An identifier that references on-line documentation for the message. Suggested usage is that *tag* includes the *label* and a unique identifying number. A sample *tag* is `"XSI:cat:146"`.

The *MSGVERB* environment variable (for message verbosity) shall determine for *fmtmsg*( ) which message components it is to select when writing messages to standard error. The value of *MSGVERB* shall be a colon-separated list of optional keywords. Valid *keywords* are: *label*, *severity*, *text*, *action*, and *tag*. If *MSGVERB* contains a keyword for a component and the component's value is not the component's null value, *fmtmsg*( ) shall include that component in the message when writing the message to standard error. If *MSGVERB* does not include a keyword for a message component, that component is shall not be included in the display of the message. The keywords may appear in any order. If *MSGVERB* is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, *fmtmsg*( ) shall select all components.

MSGVERB shall determine which components are selected for display to standard error. All message components shall be included in console messages.

RETURN VALUE

The *fmtmsg*( ) function shall return one of the following values:

MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_NOMSG	The function was unable to generate a message on standard error, but otherwise succeeded.
MM_NOCON	The function was unable to generate a console message, but otherwise succeeded.

ERRORS

None.

EXAMPLES

1. The following example of *fmtmsg*( ):

   ```
   fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
   "refer to cat in user’s reference manual", "XSI:cat:001")
   ```

   produces a complete message in the specified message format:

   ```
   XSI:cat: ERROR: illegal option
   TO FIX: refer to cat in user’s reference manual XSI:cat:001
   ```

2. When the environment variable *MSGVERB* is set as follows:

   ```
   MSGVERB=severity:text:action
   ```

   and Example 1 is used, *fmtmsg*( ) produces:

   ```
   ERROR: illegal option
   TO FIX: refer to cat in user’s reference manual
   ```

APPLICATION USAGE

One or more message components may be systematically omitted from messages generated by an application by using the null value of the argument for that component.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

printf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fmtmsg.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

fnmatch — match a filename or a pathname

SYNOPSIS

```
#include <fnmatch.h>

int fnmatch(const char *pattern, const char *string, int flags);
```

DESCRIPTION

The *fnmatch*( ) function shall match patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.1, Patterns Matching a Single Character, and Section 2.13.2, Patterns Matching Multiple Characters. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument shall modify the interpretation of *pattern* and *string*. It is the bitwise-inclusive OR of zero or more of the flags defined in **<fnmatch.h>**. If the FNM_PATHNAME flag is set in *flags*, then a slash character (`'/'`) in *string* shall be explicitly matched by a slash in *pattern*; it shall not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an ordinary character.

If FNM_NOESCAPE is not set in *flags*, a backslash character (`'\'`) in *pattern* followed by any other character shall match that second character in *string*. In particular, `"\\"` shall match a backslash in *string*. If FNM_NOESCAPE is set, a backslash character shall be treated as an ordinary character.

If FNM_PERIOD is set in *flags*, then a leading period (`'.'`) in *string* shall match a period in *pattern*; as described by rule 2 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.3, Patterns Used for Filename Expansion where the location of ‘‘leading’’ is indicated by the value of FNM_PATHNAME:

- If FNM_PATHNAME is set, a period is ‘‘leading’’ if it is the first character in *string* or if it immediately follows a slash.
- If FNM_PATHNAME is not set, a period is ‘‘leading’’ only if it is the first character of *string*.

If FNM_PERIOD is not set, then no special restrictions are placed on matching a period.

RETURN VALUE

If *string* matches the pattern specified by *pattern*, then *fnmatch*( ) shall return 0. If there is no match, *fnmatch*( ) shall return FNM_NOMATCH, which is defined in **<fnmatch.h>**. If an error occurs, *fnmatch*( ) shall return another non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *fnmatch*( ) function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The *find* utility is an example of this. It can also be used by the *pax* utility to process its *pattern* operands, or by applications that need to match strings in a similar manner.

The name *fnmatch*( ) is intended to imply *filename* match, rather than *pathname* match. The default action of this function is to match filenames, rather than pathnames, since it gives no special significance to the slash character. With the FNM_PATHNAME flag, *fnmatch*( ) does match pathnames, but without tilde expansion, parameter expansion, or special treatment for a period at the beginning of a filename.

RATIONALE

This function replaced the REG_FILENAME flag of *regcomp*( ) in early proposals of this volume of IEEE Std 1003.1-200x. It provides virtually the same functionality as the *regcomp*( ) and *regexec*( ) functions using the REG_FILENAME and REG_FSLASH flags (the REG_FSLASH flag was proposed for *regcomp*( ), and would have had the opposite effect from FNM_PATHNAME), but with a simpler function and less system overhead.

FUTURE DIRECTIONS

None.

SEE ALSO

glob( ), *wordexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fnmatch.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

NAME

fopen — open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *restrict filename, const char *restrict mode);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fopen*( ) function shall open the file whose pathname is the string pointed to by *filename*, and associates a stream with it.

The *mode* argument points to a string. If the string is one of the following, the file shall be opened in the indicated mode. Otherwise, the behavior is undefined.

r or *rb*	Open file for reading.
w or *wb*	Truncate to zero length or create file for writing.
a or *ab*	Append; open or create file for writing at end-of-file.
r+ or *rb*+ or *r*+*b*	Open file for update (reading and writing).
w+ or *wb*+ or *w*+*b*	Truncate to zero length or create file for update.
a+ or *ab*+ or *a*+*b*	Append; open or create file for update, writing at end-of-file.

CX The character `'b'` shall have no effect, but is allowed for ISO C standard conformance. Opening a file with read mode (*r* as the first character in the *mode* argument) shall fail if the file does not exist or cannot be read.

Opening a file with append mode (*a* as the first character in the *mode* argument) shall cause all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to *fseek*( ).

When a file is opened with update mode (`'+'` as the second or third character in the *mode* argument), both input and output may be performed on the associated stream. However, the application shall ensure that output is not directly followed by input without an intervening call to *fflush*( ) or to a file positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )), and input is not directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream shall be cleared.

CX If *mode* is *w*, *wb*, *a*, *ab*, *w*+, *wb*+, *w*+*b*, *a*+, *ab*+, or *a*+*b*, and the file did not previously exist, upon successful completion, *fopen*( ) function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If *mode* is *w*, *wb*, *w*+, *wb*+, or *w*+*b*, and the file did previously exist, upon successful completion, *fopen*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file. The *fopen*( ) function shall allocate a file descriptor as *open*( ) does.

XSI After a successful call to the *fopen*( ) function, the orientation of the stream shall be cleared, the encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an initial conversion state.

CX The largest value that can be represented correctly in an object of type **off_t** shall be established as the offset maximum in the open file description.

RETURN VALUE

Upon successful completion, *fopen*( ) shall return a pointer to the object controlling the stream. Otherwise, a null pointer shall be returned, CX and *errno* shall be set to indicate the error.

ERRORS

The *fopen*( ) function shall fail if:

CX	[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
CX	[EINTR]	A signal was caught during *fopen*( ).
CX	[EISDIR]	The named file is a directory and *mode* requires write access.
CX	[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
CX	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
CX	[ENAMETOOLONG]	The length of the *filename* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
CX	[ENFILE]	The maximum allowable number of files is currently open in the system.
CX	[ENOENT]	A component of *filename* does not name an existing file or *filename* is an empty string.
CX	[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.
CX	[ENOTDIR]	A component of the path prefix is not a directory.
CX	[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
CX	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type **off_t**.
CX	[EROFS]	The named file resides on a read-only file system and *mode* requires write access.

The *fopen*( ) function may fail if:

CX	[EINVAL]	The value of the *mode* argument is not valid.
CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
CX	[EMFILE]	{FOPEN_MAX} streams are currently open in the calling process.
CX	[EMFILE]	{STREAM_MAX} streams are currently open in the calling process.
CX	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

CX	[ENOMEM]	Insufficient storage space is available.
CX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *mode* requires write access.

EXAMPLES

Opening a File

The following example tries to open the file named **file** for reading. The *fopen*( ) function returns a file pointer that is used in subsequent *fgets*( ) and *fclose*( ) calls. If the program cannot open the file, it just ignores it.

```
#include <stdio.h>
...
FILE *fp;
...
void rgrep(const char *file)
{
...
    if ((fp = fopen(file, "r")) == NULL)
        return;
...
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fclose( ), *fdopen*( ), *freopen*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- The [EINVAL], [EMFILE], [ENAMETOOLONG], [ENOMEM], and [ETXTBSY] optional error conditions are added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The prototype for *fopen*( ) is updated.
- The DESCRIPTION is updated to note that if the argument *mode* points to a string other than those listed, then the behavior is undefined.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

fork — create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

The *fork*( ) function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

- The child process shall have a unique process ID.
- The child process ID also shall not match any active process group ID.
- The child process shall have a different parent process ID, which shall be the process ID of the calling process.
- The child process shall have its own copy of the parent's file descriptors. Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
- The child process shall have its own copy of the parent's open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
- XSI The child process shall have its own copy of the parent's message catalog descriptors.
- The child process' values of *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* shall be set to 0.
- The time left until an alarm clock signal shall be reset to zero, and the alarm, if any, shall be canceled; see *alarm*( ).
- XSI All *semadj* values shall be cleared.
- File locks set by the parent process shall not be inherited by the child process.
- The set of signals pending for the child process shall be initialized to the empty set.
- XSI Interval timers shall be reset in the child process.
- SEM Any semaphores that are open in the parent process shall also be open in the child process.
- ML The child process shall not inherit any address space memory locks established by the parent process via calls to *mlockall*( ) or *mlock*( ).
- MF|SHM Memory mappings created in the parent shall be retained in the child process. MAP_PRIVATE mappings inherited from the parent shall also be MAP_PRIVATE mappings in the child, and any modifications to the data in these mappings made by the parent prior to calling *fork*( ) shall be visible to the child. Any modifications to the data in MAP_PRIVATE mappings made by the parent after *fork*( ) returns shall be visible only to the parent. Modifications to the data in MAP_PRIVATE mappings made by the child shall be visible only to the child.
- PS For the SCHED_FIFO and SCHED_RR scheduling policies, the child process shall inherit the policy and priority settings of the parent process during a *fork*( ) function. For other scheduling policies, the policy and priority settings on *fork*( ) are implementation-defined.
- TMR Per-process timers created by the parent shall not be inherited by the child process.
- MSG The child process shall have its own copy of the message queue descriptors of the parent. Each of the message descriptors of the child shall refer to the same open message queue

description as the corresponding message descriptor of the parent.

- AIO No asynchronous input or asynchronous output operations shall be inherited by the child process.
- A process shall be created with a single thread. If a multi-threaded process calls *fork*( ), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the *exec* functions is called. THR Fork handlers may be established by means of the *pthread_atfork*( ) function in order to maintain application invariants across *fork*( ) calls.
- TRC TRI If the Trace option and the Trace Inherit option are both supported:

  If the calling process was being traced in a trace stream that had its inheritance policy set to POSIX_TRACE_INHERITED, the child process shall be traced into that trace stream, and the child process shall inherit the parent's mapping of trace event names to trace event type identifiers. If the trace stream in which the calling process was being traced had its inheritance policy set to POSIX_TRACE_CLOSE_FOR_CHILD, the child process shall not be traced into that trace stream. The inheritance policy is set by a call to the *posix_trace_attr_setinherited*( ) function.
- TRC If the Trace option is supported, but the Trace Inherit option is not supported:

  The child process shall not be traced into any of the trace streams of its parent process.
- TRC If the Trace option is supported, the child process of a trace controller process shall not control the trace streams controlled by its parent process.
- CPT The initial value of the CPU-time clock of the child process shall be set to zero.
- TCT The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.

All other process characteristics defined by IEEE Std 1003.1-200x shall be the same in the parent and child processes. The inheritance of process characteristics not defined by IEEE Std 1003.1-200x is unspecified by IEEE Std 1003.1-200x.

After *fork*( ), both the parent and the child processes shall be capable of executing independently before either one terminates.

RETURN VALUE

Upon successful completion, *fork*( ) shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the *fork*( ) function. Otherwise, −1 shall be returned to the parent process, no child process shall be created, and *errno* shall be set to indicate the error.

ERRORS

The *fork*( ) function shall fail if:

[EAGAIN]	The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user {CHILD_MAX} would be exceeded.

The *fork*( ) function may fail if:

[ENOMEM]	Insufficient storage space is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Many historical implementations have timing windows where a signal sent to a process group (for example, an interactive SIGINT) just prior to or during execution of *fork*( ) is delivered to the parent following the *fork*( ) but not to the child because the *fork*( ) code clears the child's set of pending signals. This volume of IEEE Std 1003.1-200x does not require, or even permit, this behavior. However, it is pragmatic to expect that problems of this nature may continue to exist in implementations that appear to conform to this volume of IEEE Std 1003.1-200x and pass available verification suites. This behavior is only a consequence of the implementation failing to make the interval between signal generation and delivery totally invisible. From the application's perspective, a *fork*( ) call should appear atomic. A signal that is generated prior to the *fork*( ) should be delivered prior to the *fork*( ). A signal sent to the process group after the *fork*( ) should be delivered to both parent and child. The implementation may actually initialize internal data structures corresponding to the child's set of pending signals to include signals sent to the process group during the *fork*( ). Since the *fork*( ) call can be considered as atomic from the application's perspective, the set would be initialized as empty and such signals would have arrived after the *fork*( ); see also **<signal.h>**.

One approach that has been suggested to address the problem of signal inheritance across *fork*( ) is to add an [EINTR] error, which would be returned when a signal is detected during the call. While this is preferable to losing signals, it was not considered an optimal solution. Although it is not recommended for this purpose, such an error would be an allowable extension for an implementation.

The [ENOMEM] error value is reserved for those implementations that detect and distinguish such a condition. This condition occurs when an implementation detects that there is not enough memory to create the process. This is intended to be returned when [EAGAIN] is inappropriate because there can never be enough memory (either primary or secondary storage) to perform the operation. Since *fork*( ) duplicates an existing process, this must be a condition where there is sufficient memory for one such process, but not for two. Many historical implementations actually return [ENOMEM] due to temporary lack of memory, a case that is not generally distinct from [EAGAIN] from the perspective of a conforming application.

Part of the reason for including the optional error [ENOMEM] is because the SVID specifies it and it should be reserved for the error condition specified there. The condition is not applicable on many implementations.

IEEE Std 1003.1-1988 neglected to require concurrent execution of the parent and child of *fork*( ). A system that single-threads processes was clearly not intended and is considered an unacceptable "toy implementation" of this volume of IEEE Std 1003.1-200x. The only objection anticipated to the phrase "executing independently" is testability, but this assertion should be testable. Such tests require that both the parent and child can block on a detectable action of the other, such as a write to a pipe or a signal. An interactive exchange of such actions should be possible for the system to conform to the intent of this volume of IEEE Std 1003.1-200x.

The [EAGAIN] error exists to warn applications that such a condition might occur. Whether it occurs or not is not in any practical sense under the control of the application because the condition is usually a consequence of the user's use of the system, not of the application's code. Thus, no application can or should rely upon its occurrence under any circumstances, nor should the exact semantics of what concept of "user" is used be of concern to the application writer. Validation writers should be cognizant of this limitation.

There are two reasons why POSIX programmers call *fork*( ). One reason is to create a new thread of control within the same program (which was originally only possible in POSIX by creating a new process); the other is to create a new process running a different program. In the latter case, the call to *fork*( ) is soon followed by a call to one of the *exec* functions.

The general problem with making *fork*( ) work in a multi-threaded world is what to do with all of the threads. There are two alternatives. One is to copy all of the threads into the new process. This causes the programmer or implementation to deal with threads that are suspended on system calls or that might be about to execute system calls that should not be executed in the new process. The other alternative is to copy only the thread that calls *fork*( ). This creates the difficulty that the state of process-local resources is usually held in process memory. If a thread that is not calling *fork*( ) holds a resource, that resource is never released in the child process because the thread whose job it is to release the resource does not exist in the child process.

When a programmer is writing a multi-threaded program, the first described use of *fork*( ), creating new threads in the same program, is provided by the *pthread_create*( ) function. The *fork*( ) function is thus used only to run new programs, and the effects of calling functions that require certain resources between the call to *fork*( ) and the call to an *exec* function are undefined.

The addition of the *forkall*( ) function to the standard was considered and rejected. The *forkall*( ) function lets all the threads in the parent be duplicated in the child. This essentially duplicates the state of the parent in the child. This allows threads in the child to continue processing and allows locks and the state to be preserved without explicit *pthread_atfork*( ) code. The calling process has to ensure that the threads processing state that is shared between the parent and child (that is, file descriptors or MAP_SHARED memory) behaves properly after *forkall*( ). For example, if a thread is reading a file descriptor in the parent when *forkall*( ) is called, then two threads (one in the parent and one in the child) are reading the file descriptor after the *forkall*( ). If this is not desired behavior, the parent process has to synchronize with such threads before calling *forkall*( ).

When *forkall*( ) is called, threads, other than the calling thread, that are in POSIX System Interfaces functions that can return with an [EINTR] error may have those functions return [EINTR] if the implementation cannot ensure that the function behaves correctly in the parent and child. In particular, *pthread_cond_wait*( ) and *pthread_cond_timedwait*( ) need to return in order to ensure that the condition has not changed. These functions can be awakened by a spurious condition wakeup rather than returning [EINTR].

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *exec*, *fcntl*( ), *posix_trace_attr_getinherited*( ), *posix_trace_trid_eventid_open*( ), *semop*( ), *signal*( ), *times*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is changed for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of *fork*( ) on a pending alarm call in the child process is clarified.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

NAME

fpathconf, pathconf — get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long fpathconf(int fildes, int name);
long pathconf(const char *path, int name);
```

DESCRIPTION

The *fpathconf*( ) and *pathconf*( ) functions shall determine the current value of a configurable limit or option (*variable*) that is associated with a file or directory.

For *pathconf*( ), the *path* argument points to the pathname of a file or directory.

For *fpathconf*( ), the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. Implementations shall support all of the variables listed in the following table and may support others. The variables in the following table come from **<limits.h>** or **<unistd.h>** and the symbolic constants, defined in **<unistd.h>**, are the corresponding values used for *name*. Support for some pathname configuration variables is dependent on implementation options (see shading and margin codes in the table below). Where an implementation option is not supported, the variable need not be supported.

	Variable	Value of *name*	Requirements
	{FILESIZEBITS}	_PC_FILESIZEBITS	3, 4
	{LINK_MAX}	_PC_LINK_MAX	1
	{MAX_CANON}	_PC_MAX_CANON	2
	{MAX_INPUT}	_PC_MAX_INPUT	2
	{NAME_MAX}	_PC_NAME_MAX	3, 4
	{PATH_MAX}	_PC_PATH_MAX	4, 5
	{PIPE_BUF}	_PC_PIPE_BUF	6
ADV	{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	
ADV	{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	
ADV	{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	
ADV	{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	
ADV	{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	
	{SYMLINK_MAX}	_PC_SYMLINK_MAX	4, 9
	_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED	7
	_POSIX_NO_TRUNC	_PC_NO_TRUNC	3, 4
	_POSIX_VDISABLE	_PC_VDISABLE	2
	_POSIX_ASYNC_IO	_PC_ASYNC_IO	8
	_POSIX_PRIO_IO	_PC_PRIO_IO	8
	_POSIX_SYNC_IO	_PC_SYNC_IO	8

Requirements

1. If *path* or *fildes* refers to a directory, the value returned shall apply to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned shall apply to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
5. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned shall apply to the referenced object. If *path* or *fildes* refers to a directory, the value returned shall apply to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
7. If *path* or *fildes* refers to a directory, the value returned shall apply to any files, other than directories, that exist or can be created within the directory.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
9. If *path* or *fildes* refers to a directory, the value returned shall be the maximum length of the string that a symbolic link in that directory can contain.

RETURN VALUE

If *name* is an invalid value, both *pathconf*( ) and *fpathconf*( ) shall return −1 and set *errno* to indicate the error.

If the variable corresponding to *name* has no limit for the *path* or file descriptor, both *pathconf*( ) and *fpathconf*( ) shall return −1 without changing *errno*. If the implementation needs to use *path* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *path*, or if the process did not have appropriate privileges to query the file specified by *path*, or *path* does not exist, *pathconf*( ) shall return −1 and set *errno* to indicate the error.

If the implementation needs to use *fildes* to determine the value of *name* and the implementation does not support the association of *name* with the file specified by *fildes*, or if *fildes* is an invalid file descriptor, *fpathconf*( ) shall return −1 and set *errno* to indicate the error.

Otherwise, *pathconf*( ) or *fpathconf*( ) shall return the current variable value for the file or directory without changing *errno*. The value returned shall not be more restrictive than the corresponding value available to the application when it was compiled with the implementation's **<limits.h>** or **<unistd.h>**.

ERRORS

The *pathconf*( ) function shall fail if:

[EINVAL]	The value of *name* is not valid.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.

The *pathconf*( ) function may fail if:

[EACCES]	Search permission is denied for a component of the path prefix.
[EINVAL]	The implementation does not support an association of the variable *name* with the specified file.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

The *fpathconf*( ) function shall fail if:

[EINVAL]	The value of *name* is not valid.

The *fpathconf*( ) function may fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The implementation does not support an association of the variable *name* with the specified file.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pathconf*( ) function was proposed immediately after the *sysconf*( ) function when it was realized that some configurable values may differ across file system, directory, or device boundaries.

For example, {NAME_MAX} frequently changes between System V and BSD-based file systems; System V uses a maximum of 14, BSD 255. On an implementation that provides both types of file systems, an application would be forced to limit all pathname components to 14 bytes, as this would be the value specified in **<limits.h>** on such a system.

Therefore, various useful values can be queried on any pathname or file descriptor, assuming that the appropriate permissions are in place.

The value returned for the variable {PATH_MAX} indicates the longest relative pathname that could be given if the specified directory is the process' current working directory. A process may not always be able to generate a name that long and use it if a subdirectory in the pathname crosses into a more restrictive file system.

The value returned for the variable _POSIX_CHOWN_RESTRICTED also applies to directories that do not have file systems mounted on them. The value may change when crossing a mount point, so applications that need to know should check for each directory. (An even easier check is to try the *chown*( ) function and look for an error in case it happens.)

Unlike the values returned by *sysconf*( ), the pathname-oriented variables are potentially more volatile and are not guaranteed to remain constant throughout the process' lifetime. For example, in between two calls to *pathconf*( ), the file system in question may have been unmounted and remounted with different characteristics.

Also note that most of the errors are optional. If one of the variables always has the same value on an implementation, the implementation need not look at *path* or *fildes* to return that value and is, therefore, not required to detect any of the errors except the meaning of [EINVAL] that indicates that the value of *name* is not valid for that variable.

If the value of any of the limits are unspecified (logically infinite), they will not be defined in **<limits.h>** and the *pathconf*( ) and *fpathconf*( ) functions return −1 without changing *errno*. This can be distinguished from the case of giving an unrecognized *name* argument because *errno* is set to [EINVAL] in this case.

Since −1 is a valid return value for the *pathconf*( ) and *fpathconf*( ) functions, applications should set *errno* to zero before calling them and check *errno* only if the return value is −1.

For the case of {SYMLINK_MAX}, since both *pathconf*( ) and *open*( ) follow symbolic links, there is no way that *path* or *fildes* could refer to a symbolic link.

FUTURE DIRECTIONS

None.

SEE ALSO

confstr( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to include {FILESIZEBITS}.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The _PC_SYMLINK_MAX entry is added to the table in the DESCRIPTION.

The *pathconf*( ) variables {POSIX_ALLOC_SIZE_MIN}, {POSIX_REC_INCR_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE}, {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_XFER_ALIGN} and their associated names are added for alignment with IEEE Std 1003.1d-1999.

NAME

fpclassify — classify real floating type

SYNOPSIS

```
#include <math.h>

int fpclassify(real-floating x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fpclassify*( ) macro shall classify its argument value as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then classification is based on the type of the argument.

RETURN VALUE

The *fpclassify*( ) macro shall return the value of the number classification macro appropriate to the value of its argument.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isfinite( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

fprintf, printf, snprintf, sprintf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int fprintf(FILE *restrict stream, const char *restrict format, ...);
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
    const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fprintf*( ) function shall place output on the named output *stream*. The *printf*( ) function shall place output on the standard output stream *stdout*. The *sprintf*( ) function shall place output followed by the null byte, `'\0'`, in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough space is available.

The *snprintf*( ) function shall be equivalent to *sprintf*( ), with the addition of the *n* argument which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to *sprintf*( ) or *snprintf*( ), the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The *format* can contain either numbered argument conversion specifications (that is, `"%n$"` and `"*m$"`), or unnumbered argument conversion specifications (that is, `%` and `*`), but not both. The only exception to this is that `%%` can be mixed with the `"%n$"` form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*–*1*)th, are specified in the format string.

In format strings containing the `"%n$"` form of conversion specification, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the `%` form of conversion specification, each argument in the argument list is used exactly once.

CX All forms of the *fprintf*( ) functions allow for the insertion of a language-dependent radix character in the output string. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`).

XSI Each conversion specification is introduced by the `'%'` character or by the character sequence `"%n$"`, after which the following appear in sequence:

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer bytes than the field width, it shall be padded with spaces by default on the left; it shall be padded on the right if the left-adjustment flag (`'−'`), described below, is given to the field width. The field width takes the form of an asterisk (`'*'`), described below, or a decimal integer.
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers; the number of digits to appear after the radix character for the `a`, `A`, `e`, `E`, `f`, and `F` conversion specifiers; the maximum number of significant digits for the `g` and `G` conversion specifiers; or the maximum number of bytes to be printed from a string in `s` XSI and `S` conversion specifiers. The precision takes the form of a period (`'.'`) followed either by an asterisk (`'*'`), described below, or an optional decimal digit string, where a null digit string is treated as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A *conversion specifier* character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (`'*'`). In this case an argument of type **int** supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a `'−'` flag followed by a positive field XSI width. A negative precision is taken as if the precision were omitted. In format strings containing the `"%n$"` form of a conversion specification, a field width or precision may be indicated by the sequence `"*m$"`, where *m* is a decimal integer in the range [1,{NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The flag characters and their meanings are:

XSI `'`	The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, or `%G`) shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.
−	The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.
+	The result of a signed conversion shall always begin with a sign (`'+'` or `'−'`). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.
<space>	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a <space> shall be prefixed to the result. This means that if the <space> and `'+'` flags both appear, the <space> flag shall be ignored.
#	Specifies that the value is to be converted to an alternative form. For `o` conversion, it increases the precision (if necessary) to force the first digit of the result to be zero. For `x`

or `X` conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For `g` and `G` conversion specifiers, trailing zeros shall *not* be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `'0'` and `'−'` flags both appear, the `'0'` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers, if a precision is specified, the `'0'` flag is ignored. XSI If the `'0'` and `'\''` flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following `n` conversion specifier applies to a pointer to a **signed char** argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **short** or **unsigned short** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short** or **unsigned short** before printing); or that a following `n` conversion specifier applies to a pointer to a **short** argument.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long** or **unsigned long** argument; that a following `n` conversion specifier applies to a pointer to a **long** argument; that a following `c` conversion specifier applies to a **wint_t** argument; that a following `s` conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell)
Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long long** or **unsigned long long** argument; or that a following `n` conversion specifier applies to a pointer to a **long long** argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following `n` conversion specifier applies to a pointer to an **intmax_t** argument.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **ptrdiff_t** or the corresponding **unsigned** type argument; or that a following `n` conversion specifier applies to a pointer to a **ptrdiff_t** argument.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d`, `i`	The **int** argument shall be converted to a signed decimal in the style `"[−]dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.
`o`	The **unsigned** argument shall be converted to unsigned octal format in the style `"dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.
`u`	The **unsigned** argument shall be converted to unsigned decimal format in the style `"dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.
`x`	The **unsigned** argument shall be converted to unsigned hexadecimal format in the style `"dddd"`; the letters `"abcdef"` are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.
`X`	Equivalent to the `x` conversion specifier, except that letters `"ABCDEF"` are used instead of `"abcdef"`.
`f`, `F`	The **double** argument shall be converted to decimal notation in the style `"[−]ddd.ddd"`, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no `'#'` flag is present, no radix character shall appear. If a radix character appears, at least one digit appears before it. The low-order digit shall be rounded in an implementation-defined manner. A **double** argument representing an infinity shall be converted in one of the styles `"[−]inf"` or `"[−]infinity"`; which style is implementation-defined. A **double** argument representing a NaN shall be converted in one of the styles `"[−]nan(n-char-sequence)"`; or `"[−]nan"` which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `"INF"`, `"INFINITY"`, or `"NAN"` instead of `"inf"`, `"infinity"`, or `"nan"`, respectively.
`e`, `E`	The **double** argument shall be converted in the style `"[−]d.ddde±dd"`, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no `'#'` flag is present, no radix character shall appear. The low-order digit shall be rounded in an implementation-defined manner. The `E` conversion specifier shall produce a number with `'E'` instead of `'e'` introducing the exponent. The exponent shall always contain at least two digits. If the value is zero, the exponent shall be zero. A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.
`g`, `G`	The **double** argument shall be converted in the style `f` or `e` (or in the style `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant

digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on the value converted; style `e` (or `E`) shall be used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros shall be removed from the fractional portion of the result; a radix character shall appear only if it is followed by a digit.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`a`, `A` A **double** argument representing a floating-point number shall be converted in the style `"[−]0x`*h.hhhh*`p±`*d*`"`, where there is one hexadecimal digit (which shall be non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the `'#'` flag is not specified, no decimal-point character shall appear. The letters `"abcdef"` shall be used for `a` conversion and the letters `"ABCDEF"` for `A` conversion. The `A` conversion specifier produces a number with `'X'` and `'P'` instead of `'x'` and `'p'`. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`c` The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall be written.

If an `l` (ell) qualifier is present, the **wint_t** argument shall be converted as if by an `ls` conversion specification with no precision and an argument that points to a two-element array of type **wchar_t**, the first element of which contains the **wint_t** argument to the `ls` conversion specification and the second element contains a null wide character.

`s` The argument shall be a pointer to an array of **char**. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

If an `l` (ell) qualifier is present, the argument shall be a pointer to an array of type **wchar_t**. Wide characters from the array shall be converted to characters (each as if by a call to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting characters shall be written up to (but not including) the terminating null character (byte). If no precision is specified, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many characters (bytes) shall be written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case shall a partial character written.

	`p`	The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.
	`n`	The argument shall be a pointer to an integer into which is written the number of bytes written to the output so far by this call to one of the *fprintf*( ) functions. No argument is converted.
XSI	`C`	Equivalent to `lc`.
XSI	`S`	Equivalent to `ls`.
	`%`	Print a `'%'` character; no argument is converted. The complete conversion specification shall be `%%`.

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by *fprintf*( ) and *printf*( ) are printed as if *fputc*( ) had been called.

For the `a` and `A` conversion specifiers, if FLT_RADIX is a power of 2, the value shall be correctly rounded to a hexadecimal floating number with the given precision.

For `a` and `A` conversions, if FLT_RADIX is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For the `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L <= D <= U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a successful execution of *fprintf*( ) or *printf*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, the *fprintf*( ) and *printf*( ) functions shall return the number of bytes transmitted.

Upon successful completion, the *sprintf*( ) function shall return the number of bytes written to *s*, excluding the terminating null byte.

Upon successful completion, the *snprintf*( ) function shall return the number of bytes that would be written to *s* had *n* been sufficiently large excluding the terminating null byte.

If an output error was encountered, these functions shall return a negative value.

If the value of *n* is zero on a call to *snprintf*( ), nothing shall be written, the number of bytes that would have been written had *n* been sufficiently large excluding the terminating null shall be returned, and *s* may be a null pointer.

ERRORS

For the conditions under which *fprintf*( ) and *printf*( ) fail and may fail, refer to *fputc*( ) or *fputwc*( ).

In addition, all forms of *fprintf*( ) may fail if:

XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.

XSI [EINVAL] There are insufficient arguments.

The *printf*( ) and *fprintf*( ) functions may fail if:

XSI [ENOMEM] Insufficient storage space is available.

The *snprintf*( ) function shall fail if:

XSI [EOVERFLOW] The value of *n* is greater than {INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than {INT_MAX}.

EXAMPLES

Printing Language-Independent Date and Time

The following statement can be used to print date and time using a language-independent format:

```
printf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the following string:

```
"%s, %s %d, %d:%.2d\n"
```

This example would produce the following message:

```
Sunday, July 3, 10:02
```

For German usage, *format* could be a pointer to the following string:

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

This definition of *format* would produce the following message:

```
Sonntag, 3. Juli, 10:02
```

Printing File Information

The following example prints information about the type, permissions, and number of links of a specific file in a directory.

The first two calls to *printf*( ) use data decoded from a previous *stat*( ) call. The user-defined *strperm*( ) function shall return a string similar to the one at the beginning of the output for the following command:

```
ls −l
```

The next call to *printf*( ) outputs the owner's name if it is found using *getpwuid*( ); the *getpwuid*( ) function shall return a **passwd** structure from which the name of the user is extracted. If the user name is not found, the program instead prints out the numeric value of the user ID.

The next call prints out the group name if it is found using *getgrgid*( ); *getgrgid*( ) is very similar to *getpwuid*( ) except that it shall return group information based on the group number. Once again, if the group is not found, the program prints the numeric value of the group for the entry.

The final call to *printf*( ) prints the size of the file.

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);
...
```

Printing a Localized Date String

The following example gets a localized date string. The *nl_langinfo*( ) function shall return the localized date string, which specifies the order and layout of the date. The *strftime*( ) function takes this information and, using the **tm** structure for values, places the date and time information into *datestring*. The *printf*( ) function then outputs *datestring* and the name of the entry.

```
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);

printf(" %s %s\n", datestring, dp->d_name);
...
```

Printing Error Information

The following example uses *fprintf*( ) to write error information to standard error.

In the first group of calls, the program tries to open the password lock file named **LOCKFILE**. If the file already exists, this is an error, as indicated by the O_EXCL flag on the *open*( ) function. If the call fails, the program assumes that someone else is updating the password file, and the program exits.

The next group of calls saves a new password file as the current password file by creating a link between **LOCKFILE** and the new password file **PASSWDFILE**.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == −1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
...
```

Printing Usage Information

The following example checks to make sure the program has the necessary arguments, and uses *fprintf*( ) to print usage information if the expected number of arguments is not present.

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options); exit(1);
}
...
```

Formatting a Decimal String

The following example prints a key and data pair on *stdout*. Note use of the `'*'` (asterisk) in the format string; this ensures the correct number of decimal places for the element based on the number of elements requested.

```
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}
```

Creating a Filename

The following example creates a filename using information from a previous *getpwnam*( ) function that returned the HOME directory of the user.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
...
char filename[PATH_MAX+1];
struct passwd *pw;
...
sprintf(filename, "%s/%d.out", pw->pw_dir, getpid());
...
```

Reporting an Event

The following example loops until an event has timed out. The *pause*( ) function waits forever unless it receives a signal. The *fprintf*( ) statement should never occur due to the possible return values of *pause*( ).

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
...
    if (pause() != −1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(errno));
}
...
```

Printing Monetary Information

The following example uses *strfmon*( ) to convert a number and store it as a formatted monetary string named *convbuf*. If the first number is printed, the program prints the format and the description; otherwise, it just prints the number.

```
#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "default formatting" },
    { "%11n", "right align within an 11 character field" },
    { "%#5n", "aligned columns for values up to 99,999" },
    { "%=*#5n", "specify a fill character" },
    { "%=0#5n", "fill characters do not use grouping" },
    { "%^#5n", "disable the grouping separator" },
    { "%^#5.0n", "round off to whole units" },
    { "%^#5.4n", "increase the precision" },
    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};
...
float input[3];
int i, j;
char convbuf[100];
...
strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s%s%s\n", table[i].format,
        convbuf, table[i].description);
}
else {
    printf("%s\n", convbuf);
}
...
```

APPLICATION USAGE

If the application calling *fprintf*( ) has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputc( ), *fscanf*( ), *setlocale*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier can now be used with `c` and `s` conversion specifiers.

The *snprintf*( ) function is new in Issue 5.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for *fprintf*( ), *printf*( ), *snprintf*( ), and *sprintf*( ) are updated, and the XSI shading is removed from *snprintf*( ).
- The description of *snprintf*( ) is aligned with the ISO C standard. Note that this supersedes the *snprintf*( ) description in The Open Group Base Resolution bwg98-006, which changed the behavior from Issue 5.
- The DESCRIPTION is updated.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

NAME

fputc — put a byte on a stream

SYNOPSIS

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fputc*( ) function shall write the byte specified by *c* (converted to an **unsigned char**) to the output stream pointed to by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and shall advance the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the byte shall be appended to the output stream.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fputc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, *fputc*( ) shall return the value it has written. Otherwise, it shall return EOF, the error indicator for the stream shall be set, CX and *errno* shall be set to indicate the error.

ERRORS

The *fputc*( ) function shall fail if either the *stream* is unbuffered or the *stream*'s buffer needs to be flushed, and:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
CX	[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for writing.
CX	[EFBIG]	An attempt was made to write to a file that exceeds the maximum file size.
XSI	[EFBIG]	An attempt was made to write to a file that exceeds the process' file size limit.
CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum.
CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
CX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
CX	[ENOSPC]	There was no free space remaining on the device containing the file.
CX	[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The *fputc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ferror( ), *fopen*( ), *getrlimit*( ), *putc*( ), *puts*( ), *setbuf*( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] and [EFBIG] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are added.

NAME

fputs — put a string on a stream

SYNOPSIS

```
#include <stdio.h>

int fputs(const char *restrict s, FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fputs*( ) function shall write the null-terminated string pointed to by *s* to the stream pointed to by *stream*. The terminating null byte shall not be written.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fputs*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, *fputs*( ) shall return a non-negative number. Otherwise, it shall return EOF, set an error indicator for the stream, CX and set *errno* to indicate the error.

ERRORS

Refer to *fputc*( ).

EXAMPLES

Printing to Standard Output

The following example gets the current time, converts it to a string using *localtime*( ) and *asctime*( ), and prints it to standard output using *fputs*( ). It then prints the number of minutes to an event for which it is waiting.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
    minutes_to_event);
...
```

APPLICATION USAGE

The *puts*( ) function appends a <newline> while *fputs*( ) does not.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *putc*( ), *puts*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *fputs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

fputwc — put a wide-character code on a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t wc, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fputwc*( ) function shall write the character corresponding to the wide-character code *wc* to the output stream pointed to by *stream*, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. If an error occurs while writing the character, the shift state of the output file is left in an undefined state.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fputwc*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, *fputwc*( ) shall return *wc*. Otherwise, it shall return WEOF, the error indicator for the stream shall be set set, CX and *errno* shall be set to indicate the error.

ERRORS

The *fputwc*( ) function shall fail if either the stream is unbuffered or data in the *stream*'s buffer needs to be written, and:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor underlying *stream* and the process would be delayed in the write operation.
CX	[EBADF]	The file descriptor underlying *stream* is not a valid file descriptor open for writing.
CX	[EFBIG]	An attempt was made to write to a file that exceeds the maximum file size or the process' file size limit.
CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
	[EILSEQ]	The wide-character code *wc* does not correspond to a valid character.
CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
CX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
CX	[ENOSPC]	There was no free space remaining on the device containing the file.
CX	[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal shall also be sent to the thread.

The *fputwc*( ) function may fail if:

CX [ENOMEM] Insufficient storage space is available.

CX [ENXIO] A request was made of a nonexistent device, or the request was outside the capabilities of the device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ferror( ), *fopen*( ), *setbuf*( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

The Optional Header (OH) marking is removed from **<stdio.h>**.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EFBIG] and [EIO] mandatory error conditions are added.
- The [ENOMEM] and [ENXIO] optional error conditions are addd.

NAME

fputws — put a wide-character string on a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int fputws(const wchar_t *restrict ws, FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fputws*( ) function shall write a character string corresponding to the (null-terminated) wide-character string pointed to by *ws* to the stream pointed to by *stream*. No character corresponding to the terminating null wide-character code shall be written.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fputws*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, *fputws*( ) shall return a non-negative number. Otherwise, it shall CX return −1, set an error indicator for the stream, and set *errno* to indicate the error.

ERRORS

Refer to *fputwc*( ).

EXAMPLES

None.

APPLICATION USAGE

The *fputws*( ) function does not append a <newline>.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *fputws*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fread — binary input

SYNOPSIS

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems,
    FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fread*( ) function shall read into the array pointed to by *ptr* up to *nitems* elements whose size is specified by *size* in bytes, from the stream pointed to by *stream*. For each object, *size* calls shall be made to the *fgetc*( ) function and the results stored, in the order read, in an array of **unsigned char** exactly overlaying the object. The file position indicator for the stream (if defined) shall be advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file position indicator for the stream is unspecified. If a partial element is read, its value is unspecified.

CX The *fread*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fgetwc*( ), *fgetws*( ), *fread*( ), *fscanf*( ), *getc*( ), *getchar*( ), *gets*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ) or *ungetwc*( ).

RETURN VALUE

Upon successful completion, *fread*( ) shall return the number of elements successfully read which is less than *nitems* only if a read error or end-of-file is encountered. If *size* or *nitems* is 0, *fread*( ) shall return 0 and the contents of the array and the state of the stream remain unchanged. CX Otherwise, if a read error occurs, the error indicator for the stream shall be set, and *errno* shall be set to indicate the error.

ERRORS

Refer to *fgetc*( ).

EXAMPLES

Reading from a Stream

The following example reads a single element from the *fp* stream into the array pointed to by *buf*.

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...
```

APPLICATION USAGE

The *ferror*( ) or *feof*( ) functions must be used to distinguish between an error condition and an end-of-file condition.

Because of possible differences in element length and byte ordering, files written using *fwrite*( ) are application-dependent, and possibly cannot be read using *fread*( ) by a different application or by the same application on a different processor.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof( ), *ferror*( ), *fgetc*( ), *fopen*( ), *getc*( ), *gets*( ), *scanf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *fread*( ) prototype is updated.
- The DESCRIPTION is updated to describe how the bytes from a call to *fgetc*( ) are stored.

NAME

free — free allocated memory

SYNOPSIS

```
#include <stdlib.h>

void free(void *ptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *free*( ) function shall cause the space pointed to by *ptr* to be deallocated; that is, made available for further allocation. If *ptr* is a null pointer, no action shall occur. Otherwise, if the ADV argument does not match a pointer earlier returned by the *calloc*( ), *malloc*( ), *posix_memalign*( ), XSI *realloc*( ), or *strdup*( ), function, or if the space has been deallocated by a call to *free*( ) or *realloc*( ), the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

RETURN VALUE

The *free*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *malloc*( ), *realloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Reference to the *valloc*( ) function is removed.

NAME

freeaddrinfo, getaddrinfo — get address information

SYNOPSIS

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
int getaddrinfo(const char *restrict nodename,
    const char *restrict servname,
    const struct addrinfo *restrict hints,
    struct addrinfo **restrict res);
```

DESCRIPTION

The *freeaddrinfo*( ) function shall free one or more **addrinfo** structures returned by *getaddrinfo*( ), along with any additional storage associated with those structures. If the *ai_next* field of the structure is not null, the entire list of structures shall be freed. The *freeaddrinfo*( ) function shall support the freeing of arbitrary sublists of an **addrinfo** list originally returned by *getaddrinfo*( ).

The *getaddrinfo*( ) function shall translate the name of a service location (for example, a host name) and/or a service name and shall return a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The *freeaddrinfo*( ) and *getaddrinfo*( ) functions shall be thread-safe.

The *nodename* and *servname* arguments are either null pointers or pointers to null-terminated strings. One or both of these two arguments shall be supplied by the application as a non-null pointer.

The format of a valid name depends on the address family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the implementation shall attempt to resolve the name in all supported families and, in absence of errors, one or more results shall be returned.

If the *nodename* argument is not null, it can be a descriptive name or can be an address string. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings using Internet standard dot notation as specified in *inet_addr*( ) are valid.

If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described in *inet_ntop*( ) are valid.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller.

If *servname* is null, the call shall return network-level addresses for the specified *nodename*. If *servname* is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the *hints* argument is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In this *hints* structure every member other than *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* shall be set to zero or a null pointer. A value of AF_UNSPEC for *ai_family* means that the caller shall accept any address family. A value of zero for *ai_socktype* means that the caller shall accept any socket type. A value of zero for *ai_protocol* means that the caller shall accept any protocol. If *hints* is a null pointer, the behavior shall be as if it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol* fields, and AF_UNSPEC for the *ai_family* field.

The *ai_flags* field to which the *hints* parameter points shall be set to zero or be the bitwise-inclusive OR of one or more of the values AI_PASSIVE, AI_CANONNAME, AI_NUMERICHOST, and AI_NUMERICSERV.

If the AI_PASSIVE flag is specified, the returned address information shall be suitable for use in binding a socket for accepting incoming connections for the specified service. In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure shall be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE flag is not specified, the returned address information shall be suitable for a call to *connect*( ) (for a connection-mode protocol) or for a call to *connect*( ), *sendto*( ), or *sendmsg*( ) (for a connectionless protocol). In this case, if the *nodename* argument is null, then the IP address portion of the socket address structure shall be set to the loopback address.

If the AI_CANONNAME flag is specified and the *nodename* argument is not null, the function shall attempt to determine the canonical name corresponding to *nodename* (for example, if *nodename* is an alias or shorthand notation for a complete name).

If the AI_NUMERICHOST flag is specified, then a non-null *nodename* string supplied shall be a numeric host address string. Otherwise, an [EAI_NONAME] error is returned. This flag shall prevent any type of name resolution service (for example, the DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, then a non-null *servname* string supplied shall be a numeric port string. Otherwise, an [EAI_NONAME] error shall be returned. This flag shall prevent any type of name resolution service (for example, NIS+) from being invoked.

IP6 If the AI_V4MAPPED flag is specified along with an *ai_family* of AF_INET6, then *getaddrinfo*( ) shall return IPv4-mapped IPv6 addresses on finding no matching IPv6 addresses (*ai_addrlen* shall be 16). The AI_V4MAPPED flag shall be ignored unless *ai_family* equals AF_INET6. If the AI_ALL flag is used with the AI_V4MAPPED flag, then *getaddrinfo*( ) shall return all matching IPv6 and IPv4 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

The *ai_socktype* field to which argument *hints* points specifies the socket type for the service, as defined in *socket*( ). If a specific socket type is not given (for example, a value of zero) and the service name could be interpreted as valid with multiple supported socket types, the implementation shall attempt to resolve the service name for all supported socket types and, in the absence of errors, all possible results shall be returned. A non-zero socket type value shall limit the returned information to values with the specified socket type.

If the *ai_family* field to which *hints* points has the value AF_UNSPEC, addresses shall be returned for use with any address family that can be used with the specified *nodename* and/or *servname*. Otherwise, addresses shall be returned for use only with the specified address family. If *ai_family* is not AF_UNSPEC and *ai_protocol* is not zero, then addresses are returned for use only with the specified address family and protocol; the value of *ai_protocol* shall be interpreted as in a call to the *socket*( ) function with the corresponding values of *ai_family* and *ai_protocol*.

RETURN VALUE

A zero return value for *getaddrinfo*( ) indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful return of *getaddrinfo*( ), the location to which *res* points shall refer to a linked list of **addrinfo** structures, each of which shall specify a socket address and information for use in creating a socket with which to use that socket address. The list shall include at least one **addrinfo** structure. The *ai_next* field of each structure contains a pointer to the next structure on the list, or a null pointer if it is the last structure on the list. Each structure on the list shall include values for use with a call to the *socket*( ) function, and a socket address for use with the *connect*( ) function or, if the AI_PASSIVE flag was specified, for use with the *bind*( ) function. The fields *ai_family*, *ai_socktype*, and *ai_protocol* shall be usable as the arguments to the *socket*( ) function to create a socket suitable for use with the returned address. The fields *ai_addr* and *ai_addrlen* are usable as the arguments to the *connect*( ) or *bind*( ) functions with such a socket, according to the AI_PASSIVE flag.

If *nodename* is not null, and if requested by the AI_CANONNAME flag, the *ai_canonname* field of the first returned **addrinfo** structure shall point to a null-terminated string containing the canonical name corresponding to the input *nodename*; if the canonical name is not available, then *ai_canonname* shall refer to the *nodename* argument or a string with the same contents. The contents of the *ai_flags* field of the returned structures are undefined.

All fields in socket address structures returned by *getaddrinfo*( ) that are not filled in through an explicit argument (for example, *sin6_flowinfo*) shall be set to zero.

Note: This makes it easier to compare socket address structures.

ERRORS

The *getaddrinfo*( ) function shall fail and return the corresponding value if:

[EAI_AGAIN]	The name could not be resolved at this time. Future attempts may succeed.
[EAI_BADFLAGS]	The *flags* parameter had an invalid value.
[EAI_FAIL]	A non-recoverable error occurred when attempting to resolve the name.
[EAI_FAMILY]	The address family was not recognized.
[EAI_MEMORY]	There was a memory allocation failure when trying to allocate storage for the return value.
[EAI_NONAME]	The name does not resolve for the supplied parameters. Neither *nodename* nor *servname* were supplied. At least one of these shall be supplied.
[EAI_SERVICE]	The service passed was not recognized for the specified socket type.
[EAI_SOCKTYPE]	The intended socket type was not recognized.
[EAI_SYSTEM]	A system error occurred; the error code can be found in *errno*.
[EAI_OVERFLOW]	An argument buffer overflowed.

EXAMPLES

None.

APPLICATION USAGE

If the caller handles only TCP and not UDP, for example, then the *ai_protocol* member of the *hints* structure should be set to IPPROTO_TCP when *getaddrinfo*( ) is called.

If the caller handles only IPv4 and not IPv6, then the *ai_family* member of the *hints* structure should be set to AF_INET when *getaddrinfo*( ) is called.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *gai_strerror*( ), *gethostbyname*( ), *getnameinfo*( ), *getservbyname*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getaddrinfo*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

freopen — open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *freopen(const char *restrict filename, const char *restrict mode,
    FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *freopen*( ) function shall first attempt to flush the stream and close any file descriptor associated with *stream*. Failure to flush or close the file descriptor successfully shall be ignored. The error and end-of-file indicators for the stream shall be cleared.

The *freopen*( ) function shall open the file whose pathname is the string pointed to by *filename* and associate the stream pointed to by *stream* with it. The *mode* argument shall be used just as in *fopen*( ).

The original stream shall be closed regardless of whether the subsequent open succeeds.

If *filename* is a null pointer, the *freopen*( ) function shall attempt to change the mode of the stream to that specified by *mode*, as if the name of the file currently associated with the stream had been used. It is implementation-defined which changes of mode are permitted (if any), and under what circumstances.

XSI After a successful call to the *freopen*( ) function, the orientation of the stream shall be cleared, the encoding rule shall be cleared, and the associated **mbstate_t** object shall be set to describe an initial conversion state.

CX The largest value that can be represented correctly in an object of type **off_t** shall be established as the offset maximum in the open file description.

RETURN VALUE

Upon successful completion, *freopen*( ) shall return the value of *stream*. Otherwise, a null pointer shall be returned, CX and *errno* shall be set to indicate the error.

ERRORS

The *freopen*( ) function shall fail if:

CX	[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *mode* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
CX	[EINTR]	A signal was caught during *freopen*( ).
CX	[EISDIR]	The named file is a directory and *mode* requires write access.
CX	[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
CX	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
CX	[ENAMETOOLONG]	The length of the *filename* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

CX	[ENFILE]	The maximum allowable number of files is currently open in the system.
CX	[ENOENT]	A component of *filename* does not name an existing file or *filename* is an empty string.
CX	[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.
CX	[ENOTDIR]	A component of the path prefix is not a directory.
CX	[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
CX	[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type **off_t**.
CX	[EROFS]	The named file resides on a read-only file system and *mode* requires write access.

The *freopen*( ) function may fail if:

CX	[EINVAL]	The value of the *mode* argument is not valid.
CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
CX	[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
CX	[ENOMEM]	Insufficient storage space is available.
CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
CX	[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *mode* requires write access.

EXAMPLES

Directing Standard Output to a File

The following example logs all standard output to the **/tmp/logfile** file.

```
#include <stdio.h>
...
FILE *fp;
...
fp = freopen ("/tmp/logfile", "a+", stdout);
...
```

APPLICATION USAGE

The *freopen*( ) function is typically used to attach the preopened *streams* associated with *stdin*, *stdout*, and *stderr* to other files.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fclose( ), *fopen*( ), *fdopen*( ), *mbsinit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate that the orientation of the stream is cleared and the conversion state of the stream is set to an initial conversion state by a successful call to the *freopen*( ) function.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [EINVAL], [ENOMEM], [ENXIO], and [ETXTBSY] optional error conditions are added.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *freopen*( ) prototype is updated.
- The DESCRIPTION is updated.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

The DESCRIPTION is updated regarding failure to close, changing the ‘‘file’’ to ‘‘file descriptor’’.

NAME

frexp, frexpf, frexpl — extract mantissa and exponent from a double precision number

SYNOPSIS

```
#include <math.h>

double frexp(double num, int *exp);
float frexpf(float num, int *exp);
long double frexpl(long double num, int *exp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall break a floating-point number *num* into a normalized fraction and an integral power of 2. The integer exponent shall be stored in the **int** object pointed to by *exp*.

RETURN VALUE

For finite arguments, these functions shall return the value *x*, such that *x* has a magnitude in the interval [½,1) or 0, and *num* equals *x* times 2 raised to the power **exp*.

MX If *num* is NaN, a NaN shall be returned, and the value of **exp* is unspecified.

If *num* is ±0, ±0 shall be returned, and the value of **exp* shall be 0.

If *num* is ±Inf, *num* shall be returned, and the value of **exp* is unspecified.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isnan( ), *ldexp*( ), *modf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *frexpf*( ) and *frexpl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

fscanf, scanf, sscanf — convert formatted input

SYNOPSIS

```
#include <stdio.h>

int fscanf(FILE *restrict stream, const char *restrict format, ... );
int scanf(const char *restrict format, ... );
int sscanf(const char *restrict s, const char *restrict format, ... );
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fscanf*( ) function shall read from the named input *stream*. The *scanf*( ) function shall read from the standard input stream *stdin*. The *sscanf*( ) function shall read from the string *s*. Each function reads bytes, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be evaluated but otherwise ignored. |

XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character `%` (see below) is replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range [1,{NL_ARGMAX}]. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages. In format strings containing the `"%n$"` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format string more than once.

The *format* can contain either form of a conversion specification—that is, `%` or `"%n$"`—but the two forms cannot be mixed within a single *format* string. The only exception to this is that `%%` or `%*` can be mixed with the `"%n$"` form. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*−1)th, are pointers. |

CX The *fscanf*( ) function in all its forms shall allow detection of a language-dependent radix character in the input string. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`). |

The format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an ordinary character (neither `'%'` nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character `'%'` XSI or the character sequence `"%n$"`, after which the following appear in sequence:

- An optional assignment-suppressing character `'*'`.
- An optional non-zero decimal integer that specifies the maximum field width.
- An option length modifier that specifies the size of the receiving object.
- A *conversion specifier* character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The *fscanf*( ) functions shall execute each directive of the format in turn. If a directive fails, as detailed below, the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters shall be executed by reading input until no more valid input can be read, or up to the first byte which is not a white-space character, which remains unread.

A directive that is an ordinary character shall be executed as follows: the next byte shall be read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent bytes shall remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification shall be executed in the following steps.

Input white-space characters (as specified by *isspace*( )) shall be skipped, unless the conversion specification includes a `[`, `c`, `C`, or `n` conversion specifier.

An item shall be read from the input, unless the conversion specification includes an `n` conversion specifier. An input item shall be defined as the longest sequence of input bytes (up to any specified maximum field width, which may be measured in characters or bytes dependent on the conversion specifier) which is an initial subsequence of a matching sequence. The first byte, if any, after the input item shall remain unread. If the length of the input item is 0, the execution of the conversion specification shall fail; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` conversion specifier, the input item (or, in the case of a `%n` conversion specification, the count of input bytes) shall be converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a `'*'`, the result of the conversion shall be placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result if the conversion specification is introduced by `%`, or in the *n*th argument if introduced by the character sequence `"%n$"`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The length modifiers and their meanings are:

`hh`	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.
`h`	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.
`l` (ell)	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long** or **unsigned long**; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **double**; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to **wchar_t**.
`ll` (ell-ell)	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long long** or **unsigned long long**.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The following conversion specifiers are valid:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of *strtol*( ) with the value 10 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **int**.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of *strtol*( ) with 0 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **int**.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of *strtoul*( ) with the value 8 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of *strtoul*( ) with the value 10 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of *strtoul*( ) with the value 16 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

`a`, `e`, `f`, `g`

Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of *strtod*( ). In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **float**.

If the *fprintf*( ) family of functions generates character string representations for infinity and NaN (a symbolic entity encoded in floating-point format) to support IEEE Std 754-1985, the *fscanf*( ) family of functions shall recognize them as input.

`s` Matches a sequence of bytes that are not white-space characters. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null character code, which shall be added automatically.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character shall be converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

`[` Matches a non-empty sequence of bytes from a set of expected bytes (the *scanset*). The normal skip over white-space characters shall be suppressed in this case. The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence and a terminating null byte, which shall be added automatically.

If an `l` (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence shall be converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent bytes in the *format* string up to and including the matching right square bracket (`']'`). The bytes between the square brackets (the *scanlist*) comprise the scanset, unless the byte after the left square bracket is a circumflex (`'^'`), in which case the scanset contains all bytes that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `"[]"` or `"[^]"`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a `'−'` is in the scanlist and is not the first character, nor the second where the first character is a `'^'`, nor the last character, the behavior is implementation-defined.

`c` Matches a sequence of bytes of the number specified by the field width (1 if no field width is present in the conversion specification). The application shall ensure that the corresponding argument is a pointer to the initial byte of an array of **char**, **signed char**, or **unsigned char** large enough to accept the sequence. No null byte is added. The normal skip over white-space characters shall be suppressed in this case.

If an `l` (ell) qualifier is present, the input shall be a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted. The application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the resulting sequence of wide characters. No null wide character is added.

`p` Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the `%p` conversion specification of the corresponding *fprintf*( ) functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the `%p` conversion specification is undefined.

`n` No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which shall be written the number of bytes read from the input so far by this call to the *fscanf*( ) functions. Execution of a `%n` conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing character or a field width, the behavior is undefined.

XSI `C` Equivalent to `lc`.

XSI `S` Equivalent to `ls`.

`%` Matches a single `'%'` character; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `A`, `E`, `F`, `G`, and `X` are also valid and shall be equivalent to `a`, `e`, `f`, `g`, and `x`, respectively.

If end-of-file is encountered during input, conversion shall be terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for `%n`) have been read (other than leading white-space characters, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure.

Reaching the end of the string in *sscanf*( ) shall be equivalent to encountering end-of-file for *fscanf*( ).

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including <newline>s) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the `%n` conversion specification.

CX The *fscanf*( ) and *scanf*( ) functions may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fread*( ), *getc*( ), *getchar*( ), *gets*( ), *fscanf*( ), or *fscanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

RETURN VALUE

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read error occurs, the error indicator for the stream is set, EOF shall be returned, CX and *errno* shall be set to indicate the error.

ERRORS

For the conditions under which the *fscanf*( ) functions fail and may fail, refer to *fgetc*( ) or *fgetwc*( ).

In addition, *fscanf*( ) may fail if:

XSI [EILSEQ] Input byte sequence does not form a valid character.

XSI [EINVAL] There are insufficient arguments.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string `"Hamster"`.

The call:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skips 0123, and places the string `"56\0"` in *name*. The next call to *getchar*( ) shall return the character `'a'`.

Reading Data into an Array

The following call uses *fscanf*( ) to read three floating-point numbers from standard input into the *input* array.

```
float input[3]; fscanf (stdin, "%f %f %f", input, input+1, input+2);
```

APPLICATION USAGE

If the application calling *fscanf*( ) has any objects of type **wint_t** or **wchar_t**, it must also include the **<wchar.h>** header to have these objects defined.

RATIONALE

This function is aligned with the ISO/IEC 9899:1999 standard, and in doing so a few ‘‘obvious’’ things were not included. Specifically, the set of characters allowed in a scanset is limited to single-byte characters. In other similar places, multi-byte characters have been permitted, but for alignment with the ISO/IEC 9899:1999 standard, it has not been done here. Applications needing this could use the corresponding wide-character functions to achieve the desired results.

FUTURE DIRECTIONS

None.

SEE ALSO

getc( ), *printf*( ), *setlocale*( ), *strtod*( ), *strtol*( ), *strtoul*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the `l` (ell) qualifier is now defined for the `c`, `s`, and `[` conversion specifiers.

The DESCRIPTION is updated to indicate that if infinity and NaN can be generated by the *fprintf*( ) family of functions, then they are recognized by the *fscanf*( ) family.

Issue 6

The Open Group Corrigendum U021/7 and U028/10 are applied. These correct several occurrences of ‘‘characters’’ in the text which have been replaced with the term ‘‘bytes’’.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for *fscanf*( ), *scanf*( ), and *sscanf*( ) are updated.
- The DESCRIPTION is updated. |
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added. |
- The `a`, `A`, and `F` conversion characters are added. |

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.

NAME

fseek, fseeko — reposition a file-position indicator in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
CX int fseeko(FILE *stream, off_t offset, int whence);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fseek*( ) function shall set the file-position indicator for the stream pointed to by *stream*. If a read or write error occurs, the error indicator for the stream shall be set and *fseek*( ) fails.

The new position, measured in bytes from the beginning of the file, shall be obtained by adding *offset* to the position specified by *whence*. The specified point is the beginning of the file for SEEK_SET, the current value of the file-position indicator for SEEK_CUR, or end-of-file for SEEK_END.

If the stream is to be used with wide-character input/output functions, the application shall ensure that *offset* is either 0 or a value returned by an earlier call to *ftell*( ) on the same stream and *whence* is SEEK_SET.

A successful call to *fseek*( ) shall clear the end-of-file indicator for the stream and undo any effects of *ungetc*( ) and *ungetwc*( ) on the same stream. After an *fseek*( ) call, the next operation on an update stream may be either input or output.

CX If the most recent operation, other than *ftell*( ), on a given stream is *fflush*( ), the file offset in the underlying open file description shall be adjusted to reflect the location specified by *fseek*( ).

The *fseek*( ) function shall allow the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The behavior of *fseek*( ) on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

If the stream is writable and buffered data had not been written to the underlying file, *fseek*( ) shall cause the unwritten data to be written to the file and shall mark the *st_ctime* and *st_mtime* fields of the file for update.

In a locale with state-dependent encoding, whether *fseek*( ) restores the stream's shift state is implementation-defined.

The *fseeko*( ) function shall be equivalent to the *fseek*( ) function except that the *offset* argument is of type **off_t**.

RETURN VALUE

CX The *fseek*( ) and *fseeko*( ) functions shall return 0 if they succeed.

CX Otherwise, they shall return −1 and set *errno* to indicate the error.

ERRORS

CX The *fseek*( ) and *fseeko*( ) functions shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed, and the call to *fseek*( ) or *fseeko*( ) causes an underlying *lseek*( ) or *write*( ) to be invoked, and:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
CX	[EBADF]	The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.
CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.
CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
CX	[EINVAL]	The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.
CX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a *write*( ) to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
CX	[ENOSPC]	There was no free space remaining on the device containing the file.
CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
CX	[EOVERFLOW]	For *fseek*( ), the resulting file offset would be a value which cannot be represented correctly in an object of type **long**.
CX	[EOVERFLOW]	For *fseeko*( ), the resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.
CX	[EPIPE]	An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal shall also be sent to the thread.
CX	[ESPIPE]	The file descriptor underlying *stream* is associated with a pipe or FIFO.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *fsetpos*( ), *ftell*( ), *getrlimit*( ), *lseek*( ), *rewind*( ), *ulimit*( ), *ungetc*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *fseeko*( ) function is added.
- The [EFBIG], [EOVERFLOW], and [ENXIO] mandatory error conditions are added.

The following change is incorporated for alignment with the FIPS requirements:

- The [EINTR] error is no longer an indication that the implementation does not report partial transfers.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The DESCRIPTION is updated to explicitly state that *fseek*( ) sets the file-position indicator, and then on error the error indicator is set and *fseek*( ) fails. This is for alignment with the ISO/IEC 9899: 1999 standard.

NAME

fsetpos — set current file position

SYNOPSIS

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fsetpos*( ) function shall set the file position and state indicators for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which the application shall ensure is a value obtained from an earlier call to *fgetpos*( ) on the same stream. If a read or write error occurs, the error indicator for the stream shall be set and *fsetpos*( ) fails.

A successful call to the *fsetpos*( ) function shall clear the end-of-file indicator for the stream and undo any effects of *ungetc*( ) on the same stream. After an *fsetpos*( ) call, the next operation on an update stream may be either input or output.

CX The behavior of *fsetpos*( ) on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

RETURN VALUE

The *fsetpos*( ) function shall return 0 if it succeeds; otherwise, it shall return a non-zero value and set *errno* to indicate the error.

ERRORS

CX The *fsetpos*( ) function shall fail if, either the *stream* is unbuffered or the *stream*'s buffer needed to be flushed, and the call to *fsetpos*( ) causes an underlying *lseek*( ) or *write*( ) to be invoked, and:

CX	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
CX	[EBADF]	The file descriptor underlying the stream file is not open for writing or the stream's buffer needed to be flushed and the file is not open.
CX	[EFBIG]	An attempt was made to write a file that exceeds the maximum file size.
XSI	[EFBIG]	An attempt was made to write a file that exceeds the process' file size limit.
CX	[EFBIG]	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
CX	[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
CX	[EINVAL]	The *whence* argument is invalid. The resulting file-position indicator would be set to a negative value.
CX	[EIO]	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a *write*( ) to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
CX	[ENOSPC]	There was no free space remaining on the device containing the file.

CX	[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
CX	[EPIPE]	The file descriptor underlying *stream* is associated with a pipe or FIFO.
CX	[EPIPE]	An attempt was made to write to a pipe or FIFO that is not open for reading by any process; a SIGPIPE signal shall also be sent to the thread.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *ftell*( ), *lseek*( ), *rewind*( ), *ungetc*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

An additional [ESPIPE] error condition is added for sockets.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The DESCRIPTION is updated to clarify that the error indicator is set for the stream on a read or write error. This is for alignment with the ISO / IEC 9899: 1999 standard.

NAME

fstat — get file status

SYNOPSIS

```
#include <sys/stat.h>

int fstat(int fildes, struct stat *buf);
```

DESCRIPTION

The *fstat*( ) function shall obtain information about an open file associated with the file descriptor *fildes*, and shall write it to the area pointed to by *buf*.

SHM If *fildes* references a shared memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

TYM If *fildes* references a typed memory object, the implementation shall update in the **stat** structure pointed to by the *buf* argument only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields, and only the S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH file permission bits need be valid. The implementation may update other fields and flags.

The *buf* argument is a pointer to a **stat** structure, as defined in **<sys/stat.h>**, into which information is placed concerning the file.

The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful values for all other file types defined in this volume of IEEE Std 1003.1-200x. The value of the member *st_nlink* shall be set to the number of links to the file.

An implementation that provides additional or alternative file access control mechanisms may, under implementation-defined conditions, cause *fstat*( ) to fail.

The *fstat*( ) function shall update any time-related fields as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update, before writing into the **stat** structure.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *fstat*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EIO]	An I/O error occurred while reading from the file system.
[EOVERFLOW]	The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The *fstat*( ) function may fail if:

[EOVERFLOW]	One of the values is too large to store into the structure pointed to by the *buf* argument.

EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure. The **/home/cnd/mod1** file is opened with read/write privileges and is passed to the open file descriptor *fildes*.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = fstat(fildes, &buffer);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EIO] mandatory error condition is added.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [EOVERFLOW] optional error condition is added.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that shared memory object semantics apply to typed memory objects.

NAME

fstatvfs, statvfs — get file system information

SYNOPSIS

XSI

```
#include <sys/statvfs.h>

int fstatvfs(int fildes, struct statvfs *buf);
int statvfs(const char *restrict path, struct statvfs *restrict buf);
```

DESCRIPTION

The *fstatvfs*( ) function shall obtain information about the file system containing the file referenced by *fildes*.

The *statvfs*( ) function shall obtain information about the file system containing the file named by *path*.

For both functions, the *buf* argument is a pointer to a **statvfs** structure that shall be filled. Read, write, or execute permission of the named file is not required.

The following flags can be returned in the *f_flag* member:

ST_RDONLY	Read-only file system.
ST_NOSUID	Setuid/setgid bits ignored by *exec*.

It is unspecified whether all members of the **statvfs** structure have meaningful values on all file systems.

RETURN VALUE

Upon successful completion, *statvfs*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *fstatvfs*( ) and *statvfs*( ) functions shall fail if:

[EIO]	An I/O error occurred while reading the file system.
[EINTR]	A signal was caught during execution of the function.
[EOVERFLOW]	One of the values to be returned cannot be represented correctly in the structure pointed to by *buf*.

The *fstatvfs*( ) function shall fail if:

[EBADF]	The *fildes* argument is not an open file descriptor.

The *statvfs*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.

The *statvfs*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Obtaining File System Information Using fstatvfs( )

The following example shows how to obtain file system information for the file system upon which the file named **/home/cnd/mod1** resides, using the *fstatvfs*( ) function. The **/home/cnd/mod1** file is opened with read/write privileges and the open file descriptor is passed to the *fstatvfs*( ) function.

```
#include <statvfs.h>
#include <fcntl.h>

struct statvfs buffer;
int            status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status  = fstatvfs(fildes, &buffer);
```

Obtaining File System Information Using statvfs( )

The following example shows how to obtain file system information for the file system upon which the file named **/home/cnd/mod1** resides, using the *statvfs*( ) function.

```
#include <statvfs.h>

struct statvfs buffer;
int            status;
...
status = statvfs("/home/cnd/mod1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *chown*( ), *creat*( ), *dup*( ), *exec*, *fcntl*( ), *link*( ), *mknod*( ), *open*( ), *pipe*( ), *read*( ), *time*( ), *unlink*( ), *utime*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/statvfs.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *statvfs*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

fsync — synchronize changes to a file

SYNOPSIS

FSC
```
#include <unistd.h>

int fsync(int fildes);
```

DESCRIPTION

The *fsync*( ) function shall request that all data for the open file descriptor named by *fildes* is to be transferred to the storage device associated with the file described by *fildes* in an implementation-defined manner. The *fsync*( ) function shall not return until the system has completed that action or until an error is detected.

SIO If _POSIX_SYNCHRONIZED_IO is defined, the *fsync*( ) function shall force all currently queued I/O operations associated with the file indicated by file descriptor *fildes* to the synchronized I/O completion state. All I/O operations shall be completed as defined for synchronized I/O file integrity completion.

RETURN VALUE

Upon successful completion, *fsync*( ) shall return 0. Otherwise, −1 shall be returned and *errno* set to indicate the error. If the *fsync*( ) function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRORS

The *fsync*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid descriptor.
[EINTR]	The *fsync*( ) function was interrupted by a signal.
[EINVAL]	The *fildes* argument does not refer to a file on which this operation is possible.
[EIO]	An I/O error occurred while reading from or writing to the file system.

In the event that any of the queued I/O operations fail, *fsync*( ) shall return the error conditions defined for *read*( ) and *write*( ).

EXAMPLES

None.

APPLICATION USAGE

The *fsync*( ) function should be used by programs which require modifications to a file to be completed before continuing; for example, a program which contains a simple transaction facility might use it to ensure that all modifications to a file or files caused by a transaction are recorded.

RATIONALE

The *fsync*( ) function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the *fsync*( ) call is recorded on the disk. Since the concepts of ‘‘buffer cache’’, ‘‘system crash’’, ‘‘physical write’’, and ‘‘non-volatile storage’’ are not defined here, the wording has to be more abstract.

If _POSIX_SYNCHRONIZED_IO is not defined, the wording relies heavily on the conformance document to tell the user what can be expected from the system. It is explicitly intended that a null implementation is permitted. This could be valid in the case where the system cannot assure non-volatile storage under any circumstances or when the system is highly fault-tolerant and the functionality is not required. In the middle ground between these extremes, *fsync*( ) might or might not actually cause data to be written where it is safe from a power failure. The conformance document should identify at least that one configuration exists (and how to obtain that configuration) where this can be assured for at least some files that the user can select to use for critical data. It is not intended that an exhaustive list is required, but rather sufficient information is provided to let the user determine that if he or she has critical data he or she can configure her system to allow it to be written to non-volatile storage.

It is reasonable to assert that the key aspects of *fsync*( ) are unreasonable to test in a test suite. That does not make the function any less valuable, just more difficult to test. A formal conformance test should probably force a system crash (power shutdown) during the test for this condition, but it needs to be done in such a way that automated testing does not require this to be done except when a formal record of the results is being made. It would also not be unreasonable to omit testing for *fsync*( ), allowing it to be treated as a quality-of-implementation issue.

FUTURE DIRECTIONS

None.

SEE ALSO

sync( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

Aligned with *fsync*( ) in the POSIX Realtime Extension. Specifically, the DESCRIPTION and RETURN VALUE sections are much expanded, and the ERRORS section is updated to indicate that *fsync*( ) can return the error conditions defined for *read*( ) and *write*( ).

Issue 6

This function is marked as part of the File Synchronization option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] and [EIO] mandatory error conditions are added.

NAME

ftell, ftello — return a file offset in a stream

SYNOPSIS

```
#include <stdio.h>

long ftell(FILE *stream);
CX off_t ftello(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ftell*( ) function shall obtain the current value of the file-position indicator for the stream pointed to by *stream*.

CX The *ftello*( ) function shall be equivalent to *ftell*( ), except that the return value is of type **off_t**. |

RETURN VALUE

CX Upon successful completion, *ftell*( ) and *ftello*( ) shall return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file.

CX Otherwise, *ftell*( ) and *ftello*( ) shall return –1, cast to **long** and **off_t** respectively, and set *errno* to indicate the error.

ERRORS

CX The *ftell*( ) and *ftello*( ) functions shall fail if:

CX	[EBADF]	The file descriptor underlying *stream* is not an open file descriptor.
CX	[EOVERFLOW]	For *ftell*( ), the current file offset cannot be represented correctly in an object of type **long**.
CX	[EOVERFLOW]	For *ftello*( ), the current file offset cannot be represented correctly in an object of type **off_t**.
CX	[ESPIPE]	The file descriptor underlying *stream* is associated with a pipe or FIFO.

The *ftell*( ) function may fail if:

CX	[ESPIPE]	The file descriptor underlying *stream* is associated with a socket.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fgetpos( ), *fopen*( ), *fseek*( ), *lseek*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *ftello*( ) function is added.
- The [EOVERFLOW] error conditions are added.

An additional [ESPIPE] error condition is added for sockets.

NAME

ftime — get date and time (**LEGACY**)

SYNOPSIS

XSI

```
#include <sys/timeb.h>

int ftime(struct timeb *tp);
```

DESCRIPTION

The *ftime*( ) function shall set the *time* and *millitm* members of the **timeb** structure pointed to by *tp* to contain the seconds and milliseconds portions, respectively, of the current time in seconds since the Epoch. The contents of the *timezone* and *dstflag* members of *tp* after a call to *ftime*( ) are unspecified.

The system clock need not have millisecond granularity. Depending on any granularity (particularly a granularity of one) renders code non-portable.

RETURN VALUE

Upon successful completion, the *ftime*( ) function shall return 0; otherwise, −1 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

Getting the Current Time and Date

The following example shows how to get the current system time values using the *ftime*( ) function. The **timeb** structure pointed to by *tp* is filled with the current system time values for *time* and *millitm*.

```
#include <sys/timeb.h>

struct timeb tp;
int          status;
...
status = ftime(&tp);
```

APPLICATION USAGE

For applications portability, the *time*( ) function should be used to determine the current time instead of *ftime*( ). Realtime applications should use *clock_gettime*( ) to determine the current time instead of *ftime*( ).

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

clock_getres( ), *ctime*( ), *gettimeofday*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/timeb.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Issue 6

This function is marked LEGACY.

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

NAME

ftok — generate an IPC key

SYNOPSIS

XSI
```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

DESCRIPTION

The *ftok*( ) function shall return a key based on *path* and *id* that is usable in subsequent calls to *msgget*( ), *semget*( ), and *shmget*( ). The application shall ensure that the *path* argument is the pathname of an existing file that the process is able to *stat*( ). |

The *ftok*( ) function shall return the same key value for all paths that name the same file, when called with the same *id* value, and return different key values when called with different *id* values or with paths that name different files existing on the same file system at the same time. It is unspecified whether *ftok*( ) shall return the same key value when called again after the file named by *path* is removed and recreated with the same name.

Only the low order 8-bits of *id* are significant. The behavior of *ftok*( ) is unspecified if these bits are 0.

RETURN VALUE

Upon successful completion, *ftok*( ) shall return a key. Otherwise, *ftok*( ) shall return (**key_t**)−1 and set *errno* to indicate the error.

ERRORS

The *ftok*( ) function shall fail if:

[EACCES] Search permission is denied for a component of the path prefix.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG]
The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of the path prefix is not a directory.

The *ftok*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

EXAMPLES

Getting an IPC Key

The following example gets a unique key that can be used by the IPC functions *semget*( ), *msgget*( ), and *shmget*( ). The key returned by *ftok*( ) for this example is based on the ID value *S* and the pathname **/tmp**. |

```
#include <sys/ipc.h>
...
key_t key;
char *path = "/tmp";
int id = 'S';

key = ftok(path, id);
```

Saving an IPC Key

The following example gets a unique key based on the pathname **/tmp** and the ID value *a*. It also assigns the value of the resulting key to the *semkey* variable so that it will be available to a later call to *semget*( ), *msgget*( ), or *shmget*( ). |

```
#include <sys/ipc.h>
...
key_t semkey;

if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}
```

APPLICATION USAGE

For maximum portability, *id* should be a single-byte character.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

msgget( ), *semget*( ), *shmget*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

ftruncate — truncate a file to a specified length

SYNOPSIS

```
#include <unistd.h>

int ftruncate(int fildes, off_t length);
```

DESCRIPTION

If *fildes* is not a valid file descriptor open for writing, the *ftruncate*( ) function shall fail.

If *fildes* refers to a regular file, the *ftruncate*( ) function shall cause the size of the file to be truncated to *length*. If the size of the file previously exceeded *length*, the extra data shall no longer be available to reads on the file. If the file previously was smaller than this size, *ftruncate*( ) shall either increase the size of the file or fail. XSI XSI-conformant systems shall increase the size of the file. If the file size is increased, the extended area shall appear as if it were zero-filled. The value of the seek pointer shall not be modified by a call to *ftruncate*( ).

Upon successful completion, if *fildes* refers to a regular file, the *ftruncate*( ) function shall mark for update the *st_ctime* and *st_mtime* fields of the file and the S_ISUID and S_ISGID bits of the file mode may be cleared. If the *ftruncate*( ) function is unsuccessful, the file is unaffected.

XSI If the request would cause the file size to exceed the soft file size limit for the process, the request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

If *fildes* refers to a directory, *ftruncate*( ) shall fail.

If *fildes* refers to any other file type, except a shared memory object, the result is unspecified.

SHM If *fildes* refers to a shared memory object, *ftruncate*( ) shall set the size of the shared memory object to *length*.

MF|SHM If the effect of *ftruncate*( ) is to decrease the size of a shared memory object or memory mapped file and whole pages beyond the new end were previously mapped, then the whole pages beyond the new end shall be discarded.

MPR If the Memory Protection option is supported, references to discarded pages shall result in the generation of a SIGBUS signal; otherwise, the result of such references is undefined.

MF|SHM If the effect of *ftruncate*( ) is to increase the size of a shared memory object, it is unspecified if the contents of any mapped pages between the old end-of-file and the new are flushed to the underlying object.

RETURN VALUE

Upon successful completion, *ftruncate*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *ftruncate*( ) function shall fail if:

[EINTR]	A signal was caught during execution.
[EINVAL]	The *length* argument was less than 0.
[EFBIG] or [EINVAL]	The *length* argument was greater than the maximum file size.
XSI [EFBIG]	The file is a regular file and *length* is greater than the offset maximum established in the open file description associated with *fildes*.
[EIO]	An I/O error occurred while reading from or writing to a file system.

[EBADF] or [EINVAL]	The *fildes* argument is not a file descriptor open for writing.
[EINVAL]	The *fildes* argument references a file that was opened without write permission.
[EROFS]	The named file resides on a read-only file system.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *ftruncate*( ) function is part of IEEE Std 1003.1-200x as it was deemed to be more useful than *truncate*( ). The *truncate*( ) function is provided as an XSI extension.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), *truncate*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE and aligned with *ftruncate*( ) in the POSIX Realtime Extension. Specifically, the DESCRIPTION is extensively reworded and [EROFS] is added to the list of mandatory errors that can be returned by *ftruncate*( ).

Large File Summit extensions are added.

Issue 6

The *truncate*( ) function has been split out into a separate reference page.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is change to indicate that if the file size is changed, and if the file is a regular file, the S_ISUID and S_ISGID bits in the file mode may be cleared.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION text is updated.

XSI-conformant systems are required to increase the size of the file if the file was previously smaller than the size requested.

NAME

ftrylockfile — stdio locking functions

SYNOPSIS

TSF

```
#include <stdio.h>

int ftrylockfile(FILE *file);
```

DESCRIPTION

Refer to *flockfile*( ).

NAME

ftw — traverse (walk) a file tree

SYNOPSIS

XSI

```
#include <ftw.h>

int ftw(const char *path, int (*fn)(const char *,
    const struct stat *ptr, int flag), int ndirs);
```

DESCRIPTION

The *ftw*( ) function shall recursively descend the directory hierarchy rooted in *path*. For each object in the hierarchy, *ftw*( ) shall call the function pointed to by *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure containing information about the object, and an integer. Possible values of the integer, defined in the **<ftw.h>** header, are:

FTW_D	For a directory.
FTW_DNR	For a directory that cannot be read.
FTW_F	For a file.
FTW_SL	For a symbolic link (but see also FTW_NS below).
FTW_NS	For an object other than a symbolic link on which *stat*( ) could not successfully be executed. If the object is a symbolic link and *stat*( ) failed, it is unspecified whether *ftw*( ) passes FTW_SL or FTW_NS to the user-supplied function.

If the integer is FTW_DNR, descendants of that directory shall not be processed. If the integer is FTW_NS, the **stat** structure contains undefined values. An example of an object that would cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with read but without execute (search) permission.

The *ftw*( ) function shall visit a directory before visiting any of its descendants.

The *ftw*( ) function shall use at most one file descriptor for each level in the tree.

The argument *ndirs* should be in the range of 1 to {OPEN_MAX}.

The tree traversal shall continue until either the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error, other than [EACCES], is detected within *ftw*( ).

The *ndirs* argument shall specify the maximum number of directory streams or file descriptors or both available for use by *ftw*( ) while traversing the tree. When *ftw*( ) returns it shall close any directory streams and file descriptors it uses not counting any opened by the application-supplied *fn* function.

The results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

The *ftw*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If the tree is exhausted, *ftw*( ) shall return 0. If the function pointed to by *fn* returns a non-zero value, *ftw*( ) shall stop its tree traversal and return whatever value was returned by the function pointed to by *fn*. If *ftw*( ) detects an error, it shall return −1 and set *errno* to indicate the error.

If *ftw*( ) encounters an error other than [EACCES] (see FTW_DNR and FTW_NS above), it shall return −1 and set *errno* to indicate the error. The external variable *errno* may contain any error value that is possible when a directory is opened or when one of the *stat* functions is executed on a directory or file.

ERRORS

The *ftw*( ) function shall fail if:

[EACCES] Search permission is denied for any component of *path* or read permission is denied for *path*.

[ELOOP] A loop exists in symbolic links encountered during resolution of the *path* argument.

[ENAMETOOLONG] The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENOENT] A component of *path* does not name an existing file or *path* is an empty string.

[ENOTDIR] A component of *path* is not a directory.

[EOVERFLOW] A field in the **stat** structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy.

The *ftw*( ) function may fail if:

[EINVAL] The value of the *ndirs* argument is invalid.

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG] Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

In addition, if the function pointed to by *fn* encounters system errors, *errno* may be set accordingly.

EXAMPLES

Walking a Directory Structure

The following example walks the current directory structure, calling the *fn* function for every directory entry, using at most 10 file descriptors:

```
#include <ftw.h>
...
if (ftw(".", fn, 10) != 0) {
    perror("ftw"); exit(2);
}
```

APPLICATION USAGE

The *ftw*( ) function may allocate dynamic storage during its operation. If *ftw*( ) is forcibly terminated, such as by *longjmp*( ) or *siglongjmp*( ) being executed by the function pointed to by *fn* or an interrupt routine, *ftw*( ) does not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function pointed to by *fn* return a non-zero value at its next invocation.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

longjmp( ), *lstat*( ), *malloc*( ), *nftw*( ), *opendir*( ), *siglongjmp*( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ftw.h>**, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

UX codings in the DESCRIPTION, RETURN VALUE, and ERRORS sections have been changed to EX.

Issue 6

The ERRORS section is updated as follows:

- The wording of the mandatory [ELOOP] error condition is updated.
- A second optional [ELOOP] error condition is added.
- The [EOVERFLOW] mandatory error condition is added.

Text is added to the DESCRIPTION to say that the *ftw*( ) function need not be reentrant and that the results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

NAME

funlockfile — stdio locking functions

SYNOPSIS

TSF

```
#include <stdio.h>

void funlockfile(FILE *file);
```

DESCRIPTION

Refer to *flockfile*( ).

NAME

fwide — set stream orientation

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *stream, int mode);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fwide*( ) function shall determine the orientation of the stream pointed to by *stream*. If *mode* is greater than zero, the function first attempts to make the stream wide-oriented. If *mode* is less than zero, the function first attempts to make the stream byte-oriented. Otherwise, *mode* is zero and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, *fwide*( ) shall not change it.

CX Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *fwide*( ), then check *errno*, and if it is non-zero, assume an error has occurred.

RETURN VALUE

The *fwide*( ) function shall return a value greater than zero if, after the call, the stream has wide-orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no orientation.

ERRORS

The *fwide*( ) function may fail if:

CX [EBADF] The *stream* argument is not a valid stream.

EXAMPLES

None.

APPLICATION USAGE

A call to *fwide*( ) with *mode* set to zero can be used to determine the current orientation of a stream.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

fwprintf, swprintf, wprintf — print formatted wide-character output

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int fwprintf(FILE *restrict stream, const wchar_t *restrict format, ...);
int swprintf(wchar_t *restrict ws, size_t n,
    const wchar_t *restrict format, ...);
int wprintf(const wchar_t *restrict format, ...);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fwprintf*( ) function shall place output on the named output *stream*. The *wprintf*( ) function shall place output on the standard output stream *stdout*. The *swprintf*( ) function shall place output followed by the null wide character in consecutive wide characters starting at **ws*; no more than *n* wide characters shall be written, including a terminating null wide character, which is always added (unless *n* is zero).

Each of these functions shall convert, format, and print its arguments under control of the *format* wide-character string. The *format* is composed of zero or more directives: *ordinary wide-characters*, which are simply copied to the output stream, and *conversion specifications*, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier wide character % (see below) is replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of *format* wide-character strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The *format* can contain either numbered argument specifications (that is, `"%n$"` and `"*m$"`), or unnumbered argument conversion specifications (that is, % and *), but not both. The only exception to this is that %% can be mixed with the `"%n$"` form. The results of mixing numbered and unnumbered argument specifications in a *format* wide-character string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the ($N-1$)th, are specified in the format wide-character string.

In *format* wide-character strings containing the `"%n$"` form of conversion specification, numbered arguments in the argument list can be referenced from the *format* wide-character string as many times as required.

In *format* wide-character strings containing the % form of conversion specification, each argument in the argument list shall be used exactly once.

CX All forms of the *fwprintf*( ) function allow for the insertion of a locale-dependent radix character in the output string, output as a wide-character value. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`).

XSI Each conversion specification is introduced by the `'%'` wide character or by the wide-character sequence `"%n$"`, after which the following appear in sequence:

- Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer wide characters than the field width, it shall be padded with spaces by default on the left; it shall be padded on the right, if the left-adjustment flag (`'−'`), described below, is given to the field width. The field width takes the form of an asterisk (`'*'`), described below, or a decimal integer.
- An optional *precision* that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers; the number of digits to appear after the radix character for the `a`, `A`, `e`, `E`, `f`, and `F` conversion specifiers; the maximum number of significant digits for the `g` and `G` conversion specifiers; or the maximum number of wide characters to be printed from a string in the `s` conversion specifiers. The precision takes the form of a period (`'.'`) followed either by an asterisk (`'*'`), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide character, the behavior is undefined.
- An optional length modifier that specifies the size of the argument.
- A *conversion specifier* wide character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (`'*'`). In this case an argument of type **int** supplies the field width or precision. Applications shall ensure that arguments specifying field width, or precision, or both appear in that order before the argument, if any, to be converted. A negative field width is taken as a `'−'` flag followed by a positive field width. A negative precision is taken as if the precision were omitted. XSI In format wide-character strings containing the `"%n$"` form of a conversion specification, a field width or precision may be indicated by the sequence `"*m$"`, where *m* is a decimal integer in the range [1,{NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The flag wide characters and their meanings are:

XSI `'`	The integer portion of the result of a decimal conversion (`%i`, `%d`, `%u`, `%f`, `%F`, `%g`, or `%G`) shall be formatted with thousands' grouping wide characters. For other conversions, the behavior is undefined. The numeric grouping wide character is used.
−	The result of the conversion shall be left-justified within the field. The conversion shall be right-justified if this flag is not specified.
+	The result of a signed conversion shall always begin with a sign (`'+'` or `'−'`). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.
<space>	If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a <space> shall be prefixed to the result. This means that if the <space> and `'+'` flags both appear, the <space> flag shall be ignored.
#	Specifies that the value is to be converted to an alternative form. For `o` conversion, it increases the precision (if necessary) to force the first digit of the result to be 0. For `x` or `X` conversion specifiers, a non-zero result shall have 0x (or 0X) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, the result shall always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For `g` and `G` conversion specifiers, trailing zeros shall *not* be removed from the result as they normally are. For other conversion

specifiers, the behavior is undefined.

`0` For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `'0'` and `'−'` flags both appear, the `'0'` flag shall be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversion specifiers, if a precision is specified, the `'0'` flag shall be ignored. If the `'0'` and `'\''` flags both appear, the grouping wide characters are inserted before zero padding. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **signed char** or **unsigned char** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **signed char** or **unsigned char** before printing); or that a following `n` conversion specifier applies to a pointer to a **signed char** argument.

`h` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **short** or **unsigned short** argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to **short** or **unsigned short** before printing); or that a following `n` conversion specifier applies to a pointer to a **short** argument.

`l` (ell) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long** or **unsigned long** argument; that a following `n` conversion specifier applies to a pointer to a **long** argument; that a following `c` conversion specifier applies to a **wint_t** argument; that a following `s` conversion specifier applies to a pointer to a **wchar_t** argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell)

Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **long long** or **unsigned long long** argument; or that a following `n` conversion specifier applies to a pointer to a **long long** argument.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an **intmax_t** or **uintmax_t** argument; or that a following `n` conversion specifier applies to a pointer to an **intmax_t** argument.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **size_t** or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to **size_t** argument.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a **ptrdiff_t** or the corresponding **unsigned** type argument; or that a following `n` conversion specifier applies to a pointer to a **ptrdiff_t** argument.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a **long double** argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d`, `i` The **int** argument shall be converted to a signed decimal in the style `"[−]`*`dddd`*`"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`o` The **unsigned** argument shall be converted to unsigned octal format in the style `"dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`u` The **unsigned** argument shall be converted to unsigned decimal format in the style `"dddd"`. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`x` The **unsigned** argument shall be converted to unsigned hexadecimal format in the style `"dddd"`; the letters `"abcdef"` are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision shall be 1. The result of converting zero with an explicit precision of zero shall be no wide characters.

`X` Equivalent to the `x` conversion specifier, except that letters `"ABCDEF"` are used instead of `"abcdef"`.

`f`, `F` The **double** argument shall be converted to decimal notation in the style `"[−]`*`ddd.ddd`*`"`, where the number of digits after the radix character shall be equal to the precision specification. If the precision is missing, it shall be taken as 6; if the precision is explicitly zero and no `'#'` flag is present, no radix character shall appear. If a radix character appears, at least one digit shall appear before it. The value shall be rounded in an implementation-defined manner to the appropriate number of digits.

A **double** argument representing an infinity shall be converted in one of the styles `"[−]inf"` or `"[−]infinity"`; which style is implementation-defined. A **double** argument representing a NaN shall be converted in one of the styles `"[−]nan"` or `"[−]nan(`*`n-char-sequence`*`)"`; which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `"INF"`, `"INFINITY"`, or `"NAN"` instead of `"inf"`, `"infinity"`, or `"nan"`, respectively.

e, *E* The **double** argument shall be converted in the style `"[−]`*`d.ddd`*e±`dd"`, where there shall be one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it shall be equal to the precision; if the precision is missing, it shall be taken as 6; if the precision is zero and no `'#'` flag is present, no radix character shall appear. The value shall be rounded in an implementation-defined manner to the appropriate number of digits. The `E` conversion wide character shall produce a number with `'E'` instead of `'e'` introducing the exponent. The exponent always shall contain at least two digits. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`g`, `G` The **double** argument shall be converted in the style `f` or `e` (or in the style `F` or `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If an explicit precision is zero, it shall be taken as 1. The style used depends on the value converted; style `e` (or `E`) shall be used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros shall be removed from the fractional portion of the result; a radix character shall appear only if it is followed by a digit.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`a`, `A` A **double** argument representing a floating-point number shlal be converted in the style `"[−]0xh.hhhhp±d"`, where there shall be one hexadecimal digit (which is non-zero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point wide character and the number of hexadecimal digits after it shall be equal to the precision; if the precision is missing and FLT_RADIX is a power of 2, then the precision shall be sufficient for an exact representation of the value; if the precision is missing and FLT_RADIX is not a power of 2, then the precision shall be sufficient to distinguish values of type **double**, except that trailing zeros may be omitted; if the precision is zero and the `'#'` flag is not specified, no decimal-point wide character shall appear. The letters `"abcdef"` are used for `a` conversion and the letters `"ABCDEF"` for `A` conversion. The `A` conversion specifier produces a number with `'X'` and `'P'` instead of `'x'` and `'p'`. The exponent shall always contain at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent shall be zero.

A **double** argument representing an infinity or NaN shall be converted in the style of an `f` or `F` conversion specifier.

`c` If no `l` (ell) qualifier is present, the **int** argument shall be converted to a wide character as if by calling the *btowc*( ) function and the resulting wide character shall be written. Otherwise, the **wint_t** argument shall be converted to **wchar_t**, and written.

`s` If no `l` (ell) qualifier is present, the application shall ensure that the argument is a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array shall be converted as if by repeated calls to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters shall be written. If the precision is not specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character.

If an `l` (ell) qualifier is present, the application shall ensure that the argument is a pointer to an array of type **wchar_t**. Wide characters from the array shall be written up to (but not including) a terminating null wide character. If no precision is specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many wide characters shall be written.

`p` The application shall ensure that the argument is a pointer to **void**. The value of the pointer shall be converted to a sequence of printable wide characters in an implementation-defined manner.

`n` The application shall ensure that the argument is a pointer to an integer into which is written the number of wide characters written to the output so far by this call to one of the *fwprintf*( ) functions. No argument shall be converted, but one shall be consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

XSI `C` Equivalent to `lc`.

XSI `S` Equivalent to `ls`.

`%` Output a `'%'` wide character; no argument shall be converted. The entire conversion specification shall be `%%`.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by *fwprintf*( ) and *wprintf*( ) shall be printed as if *fputwc*( ) had been called.

For `a` and `A` conversions, if FLT_RADIX is not a power of 2 and the result is not exactly representable in the given precision, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

For `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, if the number of significant decimal digits is at most DECIMAL_DIG, then the result should be correctly rounded. If the number of significant decimal digits is more than DECIMAL_DIG but the source value is exactly representable with DECIMAL_DIG digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having DECIMAL_DIG significant digits; the value of the resultant decimal string D should satisfy $L <= D <= U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the call to a successful execution of *fwprintf*( ) or *wprintf*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream, or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, these functions shall return the number of wide characters transmitted, excluding the terminating null wide character in the case of *swprintf*( ), or a negative value if an output error was encountered, CX and set *errno* to indicate the error.

If *n* or more wide characters were requested to be written, *swprintf*( ) shall return a negative value, CX and set *errno* to indicate the error.

ERRORS

For the conditions under which *fwprintf*( ) and *wprintf*( ) fail and may fail, refer to *fputwc*( ).

In addition, all forms of *fwprintf*( ) may fail if:

XSI [EILSEQ] A wide-character code that does not correspond to a valid character has been detected.

XSI [EINVAL] There are insufficient arguments.

In addition, *wprintf*( ) and *fwprintf*( ) may fail if:

XSI [ENOMEM] Insufficient storage space is available.

EXAMPLES

To print the language-independent date and time format, the following statement could be used:

```
wprintf(format, weekday, month, day, hour, min);
```

For American usage, *format* could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, *format* could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. Juli, 10:02
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

btowc( ), *fputwc*( ), *fwscanf*( ), *mbrtowc*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The Open Group Corrigendum U040/1 is applied to the RETURN VALUE section, describing the case if *n* or more wide characters are requested to be written using *swprintf*( ).

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for *fwprintf*( ), *swprintf*( ), and *wprintf*( ) are updated.
- The DESCRIPTION is updated.
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added.
- The `a`, `A`, and `F` conversion characters are added.
- XSI shading is removed from the description of character string representations of infinity and NaN floating-point values.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

NAME

fwrite — binary output

SYNOPSIS

```
#include <stdio.h>

size_t fwrite(const void *restrict ptr, size_t size, size_t nitems,
    FILE *restrict stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fwrite*( ) function shall write, from the array pointed to by *ptr*, up to *nitems* elements whose size is specified by *size*, to the stream pointed to by *stream*. For each object, *size* calls shall be made to the *fputc*( ) function, taking the values (in order) from an array of **unsigned char** exactly overlaying the object. The file-position indicator for the stream (if defined) shall be advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is unspecified.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *fwrite*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream, or a call to *exit*( ) or *abort*( ).

RETURN VALUE

The *fwrite*( ) function shall return the number of elements successfully written, which may be less than *nitems* if a write error is encountered. If *size* or *nitems* is 0, *fwrite*( ) shall return 0 and the state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for the stream shall be set, CX and *errno* shall be set to indicate the error.

ERRORS

Refer to *fputc*( ).

EXAMPLES

None.

APPLICATION USAGE

Because of possible differences in element length and byte ordering, files written using *fwrite*( ) are application-dependent, and possibly cannot be read using *fread*( ) by a different application or by the same application on a different processor.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ferror( ), *fopen*( ), *printf*( ), *putc*( ), *puts*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following changes are made for alignment with the ISO/IEC 9899: 1999 standard:

- The *fwrite*( ) prototype is updated.
- The DESCRIPTION is updated to clarify how the data is written out using *fputc*( ).

NAME

fwscanf, swscanf, wscanf — convert formatted wide-character input

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ... );
int swscanf(const wchar_t *restrict ws,
    const wchar_t *restrict format, ... );
int wscanf(const wchar_t *restrict format, ... );
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *fwscanf*( ) function shall read from the named input *stream*. The *wscanf*( ) function shall read from the standard input stream *stdin*. The *swscanf*( ) function shall read from the wide-character string *ws*. Each function reads wide characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored. |

XSI Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier wide character % (see below) is replaced by the sequence `"%n$"`, where *n* is a decimal integer in the range [1,{NL_ARGMAX}]. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages. In format wide-character strings containing the `"%`*n*`$"` form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format wide-character string more than once.

The *format* can contain either form of a conversion specification—that is, % or `"%`*n*`$"`— but the two forms cannot normally be mixed within a single *format* wide-character string. The only exception to this is that `%%` or `%*` can be mixed with the `"%`*n*`$"` form. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*−1)th, are pointers. |

CX The *fwscanf*( ) function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`).

The *format* is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide characters (<space>s, <tab>s, <newline>s, <vertical-tab>s, or <form-feed>s); an ordinary wide character (neither `'%'` nor a white-space character); or a conversion specification. Each conversion specification is introduced by a `'%'` XSI or the sequence `"%`*n*`$"` after which the following appear in sequence:

- An optional assignment-suppressing character `'*'`.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional length modifier that specifies the size of the receiving object.

- A conversion specifier wide character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The *fwscanf*( ) functions shall execute each directive of the format in turn. If a directive fails, as detailed below, the function shall return. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide characters is executed by reading input until no more valid input can be read, or up to the first wide character which is not a white-space wide character, which remains unread.

A directive that is an ordinary wide character shall be executed as follows. The next wide character is read from the input and compared with the wide character that comprises the directive; if the comparison shows that they are not equivalent, the directive shall fail, and the differing and subsequent wide characters remain unread. Similarly, if end-of-file, an encoding error, or a read error prevents a wide character from being read, the directive shall fail.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide character. A conversion specification is executed in the following steps.

Input white-space wide characters (as specified by *iswspace*( )) shall be skipped, unless the conversion specification includes a `[`, `c`, or `n` conversion specifier.

An item shall be read from the input, unless the conversion specification includes an `n` conversion specifier wide character. An input item is defined as the longest sequence of input wide characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide character, if any, after the input item shall remain unread. If the length of the input item is zero, the execution of the conversion specification shall fail; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` conversion specifier, the input item (or, in the case of a `%n` conversion specification, the count of input wide characters) shall be converted to a type appropriate to the conversion wide character. If the input item is not a matching sequence, the execution of the conversion specification shall fail; this condition is a matching failure. Unless assignment suppression was indicated by a `'*'`, the result of the conversion shall be placed in the object pointed to by the first argument following the *format* argument that has not already received a conversion result if the conversion specification is introduced by `%`, XSI or in the *n*th argument if introduced by the wide-character sequence `"%`*n*`$"`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The length modifiers and their meanings are:

`hh`	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **signed char** or **unsigned char**.
`h`	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **short** or **unsigned short**.
`l` (ell)	Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long** or **unsigned long**; that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **double**; or that a following `c`, `s`, or `[` conversion specifier applies to an argument with type pointer to **wchar_t**.

`ll` (ell-ell)

Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **long long** or **unsigned long long**.

`j` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **intmax_t** or **uintmax_t**.

`z` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **size_t** or the corresponding signed integer type.

`t` Specifies that a following `d`, `i`, `o`, `u`, `x`, `X`, or `n` conversion specifier applies to an argument with type pointer to **ptrdiff_t** or the corresponding **unsigned** type.

`L` Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to **long double**.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The following conversion specifier wide characters are valid:

`d` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of *wcstol*( ) with the value 10 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **int**.

`i` Matches an optionally signed integer, whose format is the same as expected for the subject sequence of *wcstol*( ) with 0 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **int**.

`o` Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of *wcstoul*( ) with the value 8 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

`u` Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of *wcstoul*( ) with the value 10 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

`x` Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of *wcstoul*( ) with the value 16 for the *base* argument. In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **unsigned**.

`a`, `e`, `f`, `g`

Matches an optionally signed floating-point number, infinity, or NaN whose format is the same as expected for the subject sequence of *wcstod*( ). In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to **float**.

If the *fwprintf*( ) family of functions generates character string representations for infinity and NaN (a symbolic entity encoded in floating-point format) to support IEEE Std 754-1985, the *fwscanf*( ) family of functions shall recognize them as input.

`s` Matches a sequence of non white-space wide characters. If no `l` (ell) qualifier is present, characters from the input field shall be converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The application shall ensure that the corresponding argument is a pointer to a character array large enough to accept the sequence and the terminating null character, which shall be added automatically.

Otherwise, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

`[` Matches a non-empty sequence of wide characters from a set of expected wide characters (the *scanset*). If no `l` (ell) qualifier is present, wide characters from the input field shall be converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The application shall ensure that the corresponding argument is a pointer to a character array large enough to accept the sequence and the terminating null character, which shall be added automatically.

If an `l` (ell) qualifier is present, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide character, which shall be added automatically.

The conversion specification includes all subsequent wide characters in the *format* string up to and including the matching right square bracket (`']'`). The wide characters between the square brackets (the *scanlist*) comprise the scanset, unless the wide character after the left square bracket is a circumflex (`'^'`), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with `"[]"` or `"[^]"`, the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise, the first right square bracket is the one that ends the conversion specification. If a `'−'` is in the scanlist and is not the first wide character, nor the second where the first wide character is a `'^'`, nor the last wide character, the behavior is implementation-defined.

`c` Matches a sequence of wide characters of exactly the number specified by the field width (1 if no field width is present in the conversion specification).

If no `l` (ell) length modifier is present, characters from the input field shall be converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialized to zero before the first wide character is converted. The corresponding argument shall be a pointer to the initial element of a character array large enough to accept the sequence. No null character is added.

If an `l` (ell) length modifier is present, the corresponding argument shall be a pointer to the initial element of an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

Otherwise, the application shall ensure that the corresponding argument is a pointer to an array of **wchar_t** large enough to accept the sequence. No null wide character is added.

`p` Matches an implementation-defined set of sequences, which shall be the same as the set of sequences that is produced by the `%p` conversion specification of the corresponding *fwprintf*( ) functions. The application shall ensure that the corresponding argument is a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise, the behavior of the %p conversion is undefined.

n No input is consumed. The application shall ensure that the corresponding argument is a pointer to the integer into which is to be written the number of wide characters read from the input so far by this call to the *fwscanf*( ) functions. Execution of a %n conversion specification shall not increment the assignment count returned at the completion of execution of the function. No argument shall be converted, but one shall be consumed. If the conversion specification includes an assignment-suppressing wide character or a field width, the behavior is undefined. |

XSI C Equivalent to lc. |

XSI S Equivalent to ls. |

% Matches a single '%' wide character; no conversion or assignment shall occur. The complete conversion specification shall be %%. |

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers A, E, F, G, and X are also valid and shall be equivalent to, respectively, a, e, f, g, and x. |

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current conversion specification (except for %n) have been read (other than leading white-space, where permitted), execution of the current conversion specification shall terminate with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) shall be terminated with an input failure. |

Reaching the end of the string in *swscanf*( ) shall be equivalent to encountering end-of-file for *fwscanf*( ). |

If conversion terminates on a conflicting input, the offending input shall be left unread in the input. Any trailing white space (including <newline>) shall be left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification. |

CX The *fwscanf*( ) and *wscanf*( ) functions may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgetwc*( ), *fgets*( ), *fgetws*( ), *fread*( ), *getc*( ), *getwc*( ), *getchar*( ), *getwchar*( ), *gets*( ), *fscanf*( ), or *fwscanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

RETURN VALUE

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read error occurs the error indicator for the stream is set, EOF shall be returned, CX and *errno* shall be set to indicate the error.

ERRORS

For the conditions under which the *fwscanf*( ) functions shall fail and may fail, refer to *fgetwc*( ).

In addition, *fwscanf*( ) may fail if:

XSI [EILSEQ]	Input byte sequence does not form a valid character.
XSI [EINVAL]	There are insufficient arguments.

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 Hamster
```

assigns to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* contains the string `"Hamster"`.

The call:

```
int i; float x; char name[50];
(void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

assigns 56 to *i*, 789.0 to *x*, skip 0123, and place the string `"56\0"` in *name*. The next call to *getchar*( ) shall return the character `'a'`.

APPLICATION USAGE

In format strings containing the `'%'` form of conversion specifications, each argument in the argument list is used exactly once.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getwc( ), *fwprintf*( ), *setlocale*( ), *wcstod*( ), *wcstol*( ), *wcstoul*( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<stdio.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The prototypes for *fwscanf*( ) and *swscanf*( ) are updated.
- The DESCRIPTION is updated. |
- The `hh`, `ll`, `j`, `t`, and `z` length modifiers are added. |
- The `a`, `A`, and `F` conversion characters are added. |

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ consistently.

NAME

gai_strerror — address and name information error description

SYNOPSIS

```
#include <netdb.h>

const char *gai_strerror(int ecode);
```

DESCRIPTION

The *gai_strerror*( ) function shall return a text string describing an error value for the *getaddrinfo*( ) and *getnameinfo*( ) functions listed in the **<netdb.h>** header.

When the *ecode* argument is one of the following values listed in the **<netdb.h>** header:

[EAI_AGAIN]
[EAI_BADFLAGS]
[EAI_FAIL]
[EAI_FAMILY]
[EAI_MEMORY]
[EAI_NONAME]
[EAI_SERVICE]
[EAI_SOCKTYPE]
[EAI_SYSTEM]

the function return value shall point to a string describing the error. If the argument is not one of those values, the function shall return a pointer to a string whose contents indicate an unknown error.

RETURN VALUE

Upon successful completion, *gai_strerror*( ) shall return a pointer to an implementation-defined string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getaddrinfo( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The Open Group Base Resolution bwg2001-009 is applied, which changes the return type from **char *** to **const char ***. This is for coordination with the IPnG Working Group.

NAME

gcvt — convert a floating-point number to a string (**LEGACY**)

SYNOPSIS

XSI

```
#include <stdlib.h>

char *gcvt(double value, int ndigit, char *buf);
```

DESCRIPTION

Refer to *ecvt*( ).

NAME

getaddrinfo — get address information

SYNOPSIS

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *restrict nodename,
    const char *restrict servname,
    const struct addrinfo *restrict hints,
    struct addrinfo **restrict res);
```

DESCRIPTION

Refer to *freeaddrinfo*( ).

NAME

getc — get a byte from a stream

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *getc*( ) function shall be equivalent to *fgetc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

RETURN VALUE

Refer to *fgetc*( ).

ERRORS

Refer to *fgetc*( ).

EXAMPLES

None.

APPLICATION USAGE

If the integer value returned by *getc*( ) is stored into a variable of type **char** and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type **char** on widening to integer is implementation-defined.

Since it may be implemented as a macro, *getc*( ) may treat incorrectly a *stream* argument with side effects. In particular, *getc*(**f*++) does not necessarily work as expected. Therefore, use of this function should be preceded by `"#undef getc"` in such situations; *fgetc*( ) could also be used. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fgetc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked — stdio with explicit client locking

SYNOPSIS

TSF

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

DESCRIPTION

Versions of the functions *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ) respectively named *getc_unlocked*( ), *getchar_unlocked*( ), *putc_unlocked*( ), and *putchar_unlocked*( ) shall be provided which are functionally equivalent to the original versions, with the exception that they are not required to be implemented in a thread-safe manner. They may only safely be used within a scope protected by *flockfile*( ) (or *ftrylockfile*( )) and *funlockfile*( ). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (**FILE ***) object, as is the case after a successful call of the *flockfile*( ) or *ftrylockfile*( ) functions.

RETURN VALUE

See *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ).

ERRORS

See *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ).

EXAMPLES

None.

APPLICATION USAGE

Since they may be implemented as macros, *getc_unlocked*( ) and *putc_unlocked*( ) may treat incorrectly a stream argument with side effects. In particular, *getc_unlocked*(*f++) and *putc_unlocked*(*f++) do not necessarily work as expected. Therefore, use of these functions in such situations should be preceded by the following statement as appropriate:

```
#undef getc_unlocked
#undef putc_unlocked
```

RATIONALE

Some I/O functions are typically implemented as macros for performance reasons (for example, *putc*( ) and *getc*( )). For safety, they need to be synchronized, but it is often too expensive to synchronize on every character. Nevertheless, it was felt that the safety concerns were more important; consequently, the *getc*( ), *getchar*( ), *putc*( ), and *putchar*( ) functions are required to be thread-safe. However, unlocked versions are also provided with names that clearly indicate the unsafe nature of their operation but can be used to exploit their higher performance. These unlocked versions can be safely used only within explicitly locked program regions, using exported locking primitives. In particular, a sequence such as:

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

is permissible, and results in the text sequence:

```
1
Line 2
```

being printed without being interspersed with output from other threads.

It would be wrong to have the standard names such as *getc*( ), *putc*( ), and so on, map to the ‘‘faster, but unsafe’’ rather than the ‘‘slower, but safe’’ versions. In either case, you would still want to inspect all uses of *getc*( ), *putc*( ), and so on, by hand when converting existing code. Choosing the safe bindings as the default, at least, results in correct code and maintains the ‘‘atomicity at the function’’ invariant. To do otherwise would introduce gratuitous synchronization errors into converted code. Other routines that modify the *stdio* (**FILE ***) structures or buffers are also safely synchronized.

Note that there is no need for functions of the form *getc_locked*( ), *putc_locked*( ), and so on, since this is the functionality of *getc*( ), *putc*( ), *et al*. It would be inappropriate to use a feature test macro to switch a macro definition of *getc*( ) between *getc_locked*( ) and *getc_unlocked*( ), since the ISO C standard requires an actual function to exist, a function whose behavior could not be changed by the feature test macro. Also, providing both the *xxx_locked*( ) and *xxx_unlocked*( ) forms leads to the confusion of whether the suffix describes the behavior of the function or the circumstances under which it should be used.

Three additional routines, *flockfile*( ), *ftrylockfile*( ), and *funlockfile*( ) (which may be macros), are provided to allow the user to delineate a sequence of I/O statements that are executed synchronously.

The *ungetc*( ) function is infrequently called relative to the other functions/macros so no unlocked variation is needed.

FUTURE DIRECTIONS

None.

SEE ALSO

getc( ), *getchar*( ), *putc*( ), *putchar*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U030/2 is applied, adding APPLICATION USAGE describing how applications should be written to avoid the case when the functions are implemented as macros.

NAME

getchar — get a byte from a stdin stream

SYNOPSIS

```
#include <stdio.h>

int getchar(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *getchar*( ) function shall be equivalent to *getc*(*stdin*).

RETURN VALUE

Refer to *fgetc*( ).

ERRORS

Refer to *fgetc*( ).

EXAMPLES

None.

APPLICATION USAGE

If the integer value returned by *getchar*( ) is stored into a variable of type **char** and then compared against the integer constant EOF, the comparison may never succeed, because sign-extension of a variable of type **char** on widening to integer is implementation-defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

getchar_unlocked — stdio with explicit client locking

SYNOPSIS

TSF

```
#include <stdio.h>

int getchar_unlocked(void);

```

DESCRIPTION

Refer to *getc_unlocked*( ).

NAME

getcontext, setcontext — get and set current user context

SYNOPSIS

XSI
```
#include <ucontext.h>

int getcontext(ucontext_t *ucp);
int setcontext(const ucontext_t *ucp);
```

DESCRIPTION

The *getcontext*( ) function shall initialize the structure pointed to by *ucp* to the current user context of the calling thread. The **ucontext_t** type that *ucp* points to defines the user context and includes the contents of the calling thread's machine registers, the signal mask, and the current execution stack.

The *setcontext*( ) function shall restore the user context pointed to by *ucp*. A successful call to *setcontext*( ) shall not return; program execution resumes at the point specified by the *ucp* argument passed to *setcontext*( ). The *ucp* argument should be created either by a prior call to *getcontext*( ) or *makecontext*( ), or by being passed as an argument to a signal handler. If the *ucp* argument was created with *getcontext*( ), program execution continues as if the corresponding call of *getcontext*( ) had just returned. If the *ucp* argument was created with *makecontext*( ), program execution continues with the function passed to *makecontext*( ). When that function returns, the thread shall continue as if after a call to *setcontext*( ) with the *ucp* argument that was input to *makecontext*( ). If the *uc_link* member of the **ucontext_t** structure pointed to by the *ucp* argument is equal to 0, then this context is the main context, and the thread shall exit when this context returns. The effects of passing a *ucp* argument obtained from any other source are unspecified.

RETURN VALUE

Upon successful completion, *setcontext*( ) shall not return and *getcontext*( ) shall return 0; otherwise, a value of –1 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

Refer to *makecontext*( ).

APPLICATION USAGE

When a signal handler is executed, the current user context is saved and a new context is created. If the thread leaves the signal handler via *longjmp*( ), then it is unspecified whether the context at the time of the corresponding *setjmp*( ) call is restored and thus whether future calls to *getcontext*( ) provide an accurate representation of the current context, since the context restored by *longjmp*( ) does not necessarily contain all the information that *setcontext*( ) requires. Signal handlers should use *siglongjmp*( ) or *setcontext*( ) instead.

Conforming applications should not modify or access the *uc_mcontext* member of **ucontext_t**. A conforming application cannot assume that context includes any process-wide static data, possibly including *errno*. Users manipulating contexts should take care to handle these explicitly when required.

Use of contexts to create alternate stacks is not defined by this volume of IEEE Std 1003.1-200x.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bsd_signal( ), *makecontext*( ), *setjmp*( ), *sigaction*( ), *sigaltstack*( ), *sigprocmask*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ucontext.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The following sentence was removed from the DESCRIPTION: ‘‘If the *ucp* argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal.’’

NAME

getcwd — get the pathname of the current working directory

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

DESCRIPTION

The *getcwd*( ) function shall place an absolute pathname of the current working directory in the array pointed to by *buf*, and return *buf*. The pathname copied to the array shall contain no components that are symbolic links. The *size* argument is the size in bytes of the character array pointed to by the *buf* argument. If *buf* is a null pointer, the behavior of *getcwd*( ) is unspecified.

RETURN VALUE

Upon successful completion, *getcwd*( ) shall return the *buf* argument. Otherwise, *getcwd*( ) shall return a null pointer and set *errno* to indicate the error. The contents of the array pointed to by *buf* are then undefined.

ERRORS

The *getcwd*( ) function shall fail if:

[EINVAL]	The *size* argument is 0.
[ERANGE]	The size argument is greater than 0, but is smaller than the length of the pathname +1.

The *getcwd*( ) function may fail if:

[EACCES]	Read or search permission was denied for a component of the pathname.
[ENOMEM]	Insufficient storage space is available.

EXAMPLES

Determining the Absolute Pathname of the Current Working Directory

The following example returns a pointer to an array that holds the absolute pathname of the current working directory. The pointer is returned in the *ptr* variable, which points to the *buf* array where the pathname is stored.

```
#include <stdlib.h>
#include <unistd.h>
...
long size;
char *buf;
char *ptr;

size = pathconf(".", _PC_PATH_MAX);

if ((buf = (char *)malloc((size_t)size)) != NULL)
    ptr = getcwd(buf, (size_t)size);
...
```

APPLICATION USAGE

None.

RATIONALE

Since the maximum pathname length is arbitrary unless {PATH_MAX} is defined, an application generally cannot supply a *buf* with *size* {{PATH_MAX}+1}.

Having *getcwd*( ) take no arguments and instead use the *malloc*( ) function to produce space for the returned argument was considered. The advantage is that *getcwd*( ) knows how big the working directory pathname is and can allocate an appropriate amount of space. But the programmer would have to use the *free*( ) function to free the resulting object, or each use of *getcwd*( ) would further reduce the available memory. Also, *malloc*( ) and *free*( ) are used nowhere else in this volume of IEEE Std 1003.1-200x. Finally, *getcwd*( ) is taken from the SVID where it has the two arguments used in this volume of IEEE Std 1003.1-200x.

The older function *getwd*( ) was rejected for use in this context because it had only a buffer argument and no *size* argument, and thus had no way to prevent overwriting the buffer, except to depend on the programmer to provide a large enough buffer.

On some implementations, if *buf* is a null pointer, *getcwd*( ) may obtain *size* bytes of memory using *malloc*( ). In this case, the pointer returned by *getcwd*( ) may be used as the argument in a subsequent call to *free*( ). Invoking *getcwd*( ) with *buf* as a null pointer is not recommended in conforming applications.

If a program is operating in a directory where some (grand)parent directory does not permit reading, *getcwd*( ) may fail, as in most implementations it must read the directory to determine the name of the file. This can occur if search, but not read, permission is granted in an intermediate directory, or if the program is placed in that directory by some more privileged process (for example, login). Including the [EACCES] error condition makes the reporting of the error consistent and warns the application writer that *getcwd*( ) can fail for reasons beyond the control of the application writer or user. Some implementations can avoid this occurrence (for example, by implementing *getcwd*( ) using *pwd*, where *pwd* is a set-user-root process), thus the error was made optional. Since this volume of IEEE Std 1003.1-200x permits the addition of other errors, this would be a common addition and yet one that applications could not be expected to deal with without this addition.

FUTURE DIRECTIONS

None.

SEE ALSO

malloc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [ENOMEM] optional error condition is added.

NAME

getdate — convert user format date and time

SYNOPSIS

XSI

```
#include <time.h>

struct tm *getdate(const char *string);
```

DESCRIPTION

The *getdate*( ) function shall convert a string representation of a date or time into a broken-down time.

The external variable or macro *getdate_err* is used by *getdate*( ) to return error values.

Templates are used to parse and interpret the input string. The templates are contained in a text file identified by the environment variable *DATEMSK*. The *DATEMSK* variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format.

The following conversion specifications shall be supported:

`%%`	Equivalent to `%`.
`%a`	Abbreviated weekday name.
`%A`	Full weekday name.
`%b`	Abbreviated month name.
`%B`	Full month name.
`%c`	Locale's appropriate date and time representation.
`%C`	Century number [00,99]; leading zeros are permitted but not required.
`%d`	Day of month [01,31]; the leading 0 is optional.
`%D`	Date as `%m`/`%d`/`%y`.
`%e`	Equivalent to `%d`.
`%h`	Abbreviated month name.
`%H`	Hour [00,23].
`%I`	Hour [01,12].
`%m`	Month number [01,12].
`%M`	Minute [00,59].
`%n`	Equivalent to <newline>.
`%p`	Locale's equivalent of either AM or PM.
`%r`	The locale's appropriate representation of time in AM and PM notation. In the POSIX locale, this shall be equivalent to `%I:%M:%S %p`.
`%R`	Time as `%H:%M`.
`%S`	Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap second data must come from some external source.

`%t`	Equivalent to <tab>.
`%T`	Time as `%H:%M:%S`.
`%w`	Weekday number (Sunday = [0,6]).
`%x`	Locale's appropriate date representation.
`%X`	Locale's appropriate time representation.
`%y`	Year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive. **Note:** It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
`%Y`	Year as `"ccyy"` (for example, 2001).
`%Z`	Timezone name or no characters if no timezone exists. If the timezone supplied by `%Z` is not the timezone that *getdate*( ) expects, an invalid input specification error shall result. The *getdate*( ) function calculates an expected timezone based on information supplied to the function (such as the hour, day, and month).

The match between the template and input specification performed by *getdate*( ) shall be case-insensitive.

The month and weekday names can consist of any combination of upper and lowercase letters. The process can request that the input date or time specification be in a specific language by setting the *LC_TIME* category (see *setlocale*( )).

Leading 0s are not necessary for the descriptors that allow leading 0s. However, at most two digits are allowed for those descriptors, including leading 0s. Extra whitespace in either the template file or in *string* shall be ignored.

The results are undefined if the conversion specifications `%c`, `%x`, and `%X` include unsupported conversion specifications.

The following rules apply for converting the input specification into the internal format:

- If `%Z` is being scanned, then *getdate*( ) shall initialize the broken-down time to be the current time in the scanned timezone. Otherwise, it shall initialize the broken-down time based on the current local time as if *localtime*( ) had been called.
- If only the weekday is given, the day chosen shall be the day, starting with today and moving into the future, which first matches the named day.
- If only the month (and no year) is given, the month chosen shall be the month, starting with the current month and moving into the future, which first matches the named month. The first day of the month shall be assumed if no day is given.
- If no hour, minute, and second are given the current hour, minute, and second shall be assumed.
- If no date is given, the hour chosen shall be the hour, starting with the current hour and moving into the future, which first matches the named hour.

If a conversion specification in the DATEMSK file does not correspond to one of the conversion specifications above, the behavior is unspecified.

The *getdate*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getdate*( ) shall return a pointer to a **struct tm**. Otherwise, it shall return a null pointer and set *getdate_err* to indicate the error.

ERRORS

The *getdate*( ) function shall fail in the following cases, setting *getdate_err* to the value shown in the list below. Any changes to *errno* are unspecified.

1. The *DATEMSK* environment variable is null or undefined.
2. The template file cannot be opened for reading.
3. Failed to get file status information.
4. The template file is not a regular file.
5. An I/O error is encountered while reading the template file.
6. Memory allocation failed (not enough memory available).
7. There is no line in the template that matches the input.
8. Invalid input specification. For example, February 31; or a time is specified that cannot be represented in a **time_t** (representing the time in seconds since the Epoch).

EXAMPLES

1. The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d,%m,%Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p,%B %dnd
%A den %d. %B %Y %H.%M Uhr
```

2. The following are examples of valid input specifications for the template in Example 1:

```
getdate("10/1/87 4 PM");
getdate("Friday");
getdate("Friday September 18, 1987, 10:30:30");
getdate("24,9,1986 10:30");
getdate("at monday the 1st of december in 1986");
getdate("run job at 3 PM, december 2nd");
```

If the *LC_TIME* category is set to a German locale that includes *freitag* as a weekday name and *oktober* as a month name, the following would be valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr");
```

3. The following example shows how local date and time specification can be defined in the template:

Invocation	Line in Template
`getdate("11/27/86")`	`%m/%d/%y`
`getdate("27.11.86")`	`%d.%m.%y`
`getdate("86-11-27")`	`%y-%m-%d`
`getdate("Friday 12:00:00")`	`%A %H:%M:%S`

4. The following examples help to illustrate the above rules assuming that the current date is Mon Sep 22 12:19:47 EDT 1986 and the *LC_TIME* category is set to the default C locale:

Input	Line in Template	Date
`Mon`	`%a`	Mon Sep 22 12:19:47 EDT 1986
`Sun`	`%a`	Sun Sep 28 12:19:47 EDT 1986
`Fri`	`%a`	Fri Sep 26 12:19:47 EDT 1986
`September`	`%B`	Mon Sep 1 12:19:47 EDT 1986
`January`	`%B`	Thu Jan 1 12:19:47 EST 1987
`December`	`%B`	Mon Dec 1 12:19:47 EST 1986
`Sep Mon`	`%b %a`	Mon Sep 1 12:19:47 EDT 1986
`Jan Fri`	`%b %a`	Fri Jan 2 12:19:47 EST 1987
`Dec Mon`	`%b %a`	Mon Dec 1 12:19:47 EST 1986
`Jan Wed 1989`	`%b %a %Y`	Wed Jan 4 12:19:47 EST 1989
`Fri 9`	`%a %H`	Fri Sep 26 09:00:00 EDT 1986
`Feb 10:30`	`%b %H:%S`	Sun Feb 1 10:00:30 EST 1987
`10:30`	`%H:%M`	Tue Sep 23 10:30:00 EDT 1986
`13:30`	`%H:%M`	Mon Sep 22 13:30:00 EDT 1986

APPLICATION USAGE

Although historical versions of *getdate*( ) did not require that **<time.h>** declare the external variable *getdate_err*, this volume of IEEE Std 1003.1-200x does require it. The standard developers encourage applications to remove declarations of *getdate_err* and instead incorporate the declaration by including **<time.h>**. |

Applications should use `%Y` (4-digit years) in preference to `%y` (2-digit years).

RATIONALE

In standard locales, the conversion specifications `%c`, `%x`, and `%X` do not include unsupported conversion specifiers and so the text regarding results being undefined is not a problem in that case. |

FUTURE DIRECTIONS

None.

SEE ALSO

ctime( ), *localtime*( ), *setlocale*( ), *strftime*( ), *times*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The last paragraph of the DESCRIPTION is added.

The `%C` conversion specification is added, and the exact meaning of the `%y` conversion specification is clarified in the DESCRIPTION.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The `%R` conversion specifications is changed to follow historical practice.

Issue 6

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

The description of `%S` is updated so that the valid range [00,60] rather than [00,61]. |

The DESCRIPTION is updated to refer to conversion specifications instead of field descriptors for consistency with other functions. |

NAME

getegid — get the effective group ID

SYNOPSIS

```
#include <unistd.h>

gid_t getegid(void);
```

DESCRIPTION

The *getegid*( ) function shall return the effective group ID of the calling process.

RETURN VALUE

The *getegid*( ) function shall always be successful and no return value is reserved to indicate an error. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

geteuid( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getenv — get value of an environment variable

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *getenv*( ) function shall search the environment of the calling process (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables) for the environment variable *name* if it exists and return a pointer to the value of the environment variable. If the specified environment variable cannot be found, a null pointer shall be returned. The application shall ensure that it does not modify the string pointed to by the *getenv*( ) function.

CX The string pointed to may be overwritten by a subsequent call to *getenv*( ), *setenv*( ), or *unsetenv*( ), but shall not be overwritten by a call to any other function in this volume of IEEE Std 1003.1-200x.

CX If the application modifies *environ* or the pointers to which it points, the behavior of *getenv*( ) is undefined. |

The *getenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *getenv*( ) shall return a pointer to a string containing the *value* for the specified *name*. If the specified name cannot be found in the environment of the calling process, a null pointer shall be returned.

The return value from *getenv*( ) may point to static data which may be overwritten by CX subsequent calls to *getenv*( ), *setenv*( ), or *unsetenv*( ).

XSI On XSI-conformant systems, the return value from *getenv*( ) may point to static data which may also be overwritten by subsequent calls to *putenv*( ).

ERRORS

No errors are defined.

EXAMPLES

Getting the Value of an Environment Variable

The following example gets the value of the *HOME* environment variable.

```
#include <stdlib.h>
...
const char *name = "HOME";
char *value;

value = getenv(name);
```

APPLICATION USAGE

None.

RATIONALE

The *clearenv*( ) function was considered but rejected. The *putenv*( ) function has now been included for alignment with the Single UNIX Specification.

The *getenv*( ) function is inherently not reentrant because it returns a value pointing to static data.

Conforming applications are required not to modify *environ* directly, but to use only the functions described here to manipulate the process environment as an abstract object. Thus, the implementation of the environment access functions has complete control over the data structure used to represent the environment (subject to the requirement that *environ* be maintained as a list of strings with embedded equal signs for applications that wish to scan the environment). This constraint allows the implementation to properly manage the memory it allocates, either by using allocated storage for all variables (copying them on the first invocation of *setenv*( ) or *unsetenv*( )), or keeping track of which strings are currently in allocated space and which are not, via a separate table or some other means. This enables the implementation to free any allocated space used by strings (and perhaps the pointers to them) stored in *environ* when *unsetenv*( ) is called. A C runtime start-up procedure (that which invokes *main*( ) and perhaps initializes *environ*) can also initialize a flag indicating that none of the environment has yet been copied to allocated storage, or that the separate table has not yet been initialized.

In fact, for higher performance of *getenv*( ), the implementation could also maintain a separate copy of the environment in a data structure that could be searched much more quickly (such as an indexed hash table, or a binary tree), and update both it and the linear list at *environ* when *setenv*( ) or *unsetenv*( ) is invoked.

Performance of *getenv*( ) can be important for applications which have large numbers of environment variables. Typically, applications like this use the environment as a resource | database of user-configurable parameters. The fact that these variables are in the user's shell | environmentt usually means that any other program that uses environment variables (such as *ls*, which attempts to use *COLUMNS*, or really almost any utility (*LANG*, *LC_ALL*, and so on) is similarly slowed down by the linear search through the variables.

An implementation that maintains separate data structures, or even one that manages the memory it consumes, is not currently required as it was thought it would reduce consensus among implementors who do not want to change their historical implementations.

The POSIX Threads Extension states that multi-threaded applications must not modify *environ* directly, and that IEEE Std 1003.1-200x is providing functions which such applications can use in the future to manipulate the environment in a thread-safe manner. Thus, moving away from application use of *environ* is desirable from that standpoint as well.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *putenv*( ), *setenv*( ), *unsetenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, | **<stdlib.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- References added to the new *setenv*( ) and *unsetenv*( ) functions.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

geteuid — get the effective user ID

SYNOPSIS

```
#include <unistd.h>

uid_t geteuid(void);
```

DESCRIPTION

The *geteuid*( ) function shall return the effective user ID of the calling process.

RETURN VALUE

The *geteuid*( ) function shall always be successful and no return value is reserved to indicate an error. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getgid — get the real group ID

SYNOPSIS

```
#include <unistd.h>

gid_t getgid(void);
```

DESCRIPTION

The *getgid*( ) function shall return the real group ID of the calling process.

RETURN VALUE

The *getgid*( ) function shall always be successful and no return value is reserved to indicate an error. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid( ), *geteuid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getgrent — get the group database entry

SYNOPSIS

XSI

```
#include <grp.h>

struct group *getgrent(void);
```

DESCRIPTION

Refer to *endgrent*( ).

NAME

getgrgid, getgrgid_r — get group database entry for a group ID

SYNOPSIS

```
#include <grp.h>

struct group *getgrgid(gid_t gid);
int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
    size_t bufsize, struct group **result);
```

TSF

DESCRIPTION

The *getgrgid*( ) function shall search the group database for an entry with a matching *gid*.

The *getgrgid*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *getgrgid_r*( ) function shall update the **group** structure pointed to by *grp* and store a pointer to that structure at the location pointed to by *result*. The structure shall contain an entry from the group database with a matching *gid*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on error or if the requested entry is not found.

RETURN VALUE

Upon successful completion, *getgrgid*( ) shall return a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The *getgrgid*( ) function shall return a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

TSF If successful, the *getgrgid_r*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *getgrgid*( ) and *getgrgid_r*( ) functions may fail if:

[EIO]	An I/O error has occurred.
[EINTR]	A signal was caught during *getgrgid*( ).
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

TSF The *getgrgid_r*( ) function may fail if:

[ERANGE]	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

EXAMPLES

Finding an Entry in the Group Database

The following example uses *getgrgid*( ) to search the group database for a group ID that was previously stored in a **stat** structure, then prints out the group name if it is found. If the group is not found, the program prints the numeric value of the group for the entry.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
...
struct stat statbuf;
struct group *grp;
...
if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8d", statbuf.st_gid);
...
```

APPLICATION USAGE

Applications wishing to check for error situations should set *errno* to 0 before calling *getgrgid*( ). If *errno* is set on return, an error occurred.

The *getgrgid_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent( ), *getgrnam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getgrgid_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getgrgid*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getgrgid_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *gid*.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

NAME

getgrnam, getgrnam_r — search group database for a name

SYNOPSIS

```
#include <grp.h>

struct group *getgrnam(const char *name);
int getgrnam_r(const char *name, struct group *grp, char *buffer,
    size_t bufsize, struct group **result);
```

TSF (applies to `getgrnam_r`)

DESCRIPTION

The *getgrnam*( ) function shall search the group database for an entry with a matching *name*.

The *getgrnam*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *getgrnam_r*( ) function shall update the **group** structure pointed to by *grp* and store a pointer to that structure at the location pointed to by *result*. The structure shall contain an entry from the group database with a matching *gid* or *name*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf*( ) parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

RETURN VALUE

The *getgrnam*( ) function shall return a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The *getgrnam*( ) function shall return a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* shall be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to *getgrent*( ), *getgrgid*( ), or *getgrnam*( ).

TSF If successful, the *getgrnam_r*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *getgrnam*( ) and *getgrnam_r*( ) functions may fail if:

[EIO]	An I/O error has occurred.
[EINTR]	A signal was caught during *getgrnam*( ).
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

The *getgrnam_r*( ) function may fail if:

TSF [ERANGE]	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

EXAMPLES

None.

APPLICATION USAGE

Applications wishing to check for error situations should set *errno* to 0 before calling *getgrnam*( ). If *errno* is set on return, an error occurred.

The *getgrnam_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endgrent( ), *getgrgid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<grp.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getgrnam_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getgrnam*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getgrnam_r*( ) function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

NAME

getgroups — get supplementary group IDs

SYNOPSIS

```
#include <unistd.h>

int getgroups(int gidsetsize, gid_t grouplist[]);
```

DESCRIPTION

The *getgroups*( ) function shall fill in the array *grouplist* with the current supplementary group IDs of the calling process. It is implementation-defined whether *getgroups*( ) also returns the effective group ID in the *grouplist* array. |

The *gidsetsize* argument specifies the number of elements in the array *grouplist*. The actual number of group IDs stored in the array shall be returned. The values of array entries with indices greater than or equal to the value returned are undefined. |

If *gidsetsize* is 0, *getgroups*( ) shall return the number of group IDs that it would otherwise return without modifying the array pointed to by *grouplist*.

If the effective group ID of the process is returned with the supplementary group IDs, the value returned shall always be greater than or equal to one and less than or equal to the value of {NGROUPS_MAX}+1.

RETURN VALUE

Upon successful completion, the number of supplementary group IDs shall be returned. A return value of –1 indicates failure and *errno* shall be set to indicate the error.

ERRORS

The *getgroups*( ) function shall fail if:

[EINVAL]	The *gidsetsize* argument is non-zero and less than the number of group IDs that would have been returned.

EXAMPLES

Getting the Supplementary Group IDs of the Calling Process

The following example places the current supplementary group IDs of the calling process into the *group* array.

```
#include <sys/types.h>
#include <unistd.h>
...
gid_t *group;
int nogroups;
long ngroups_max;

ngroups_max = sysconf(_SC_NGROUPS_MAX) + 1;
group = (gid_t *)malloc(ngroups_max *sizeof(gid_t));

ngroups = getgroups(ngroups_max, group);
```

APPLICATION USAGE

None.

RATIONALE

The related function *setgroups*( ) is a privileged operation and therefore is not covered by this volume of IEEE Std 1003.1-200x.

As implied by the definition of supplementary groups, the effective group ID may appear in the array returned by *getgroups*( ) or it may be returned only by *getegid*( ). Duplication may exist, but the application needs to call *getegid*( ) to be sure of getting all of the information. Various implementation variations and administrative sequences cause the set of groups appearing in the result of *getgroups*( ) to vary in order and as to whether the effective group ID is included, even when the set of groups is the same (in the mathematical sense of ‘‘set’’). (The history of a process and its parents could affect the details of result.)

Applications writers should note that {NGROUPS_MAX} is not necessarily a constant on all implementations.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid( ), *setgid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the DESCRIPTION.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- A return value of 0 is not permitted, because {NGROUPS_MAX} cannot be 0. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Explanation added that the effective group ID may be included in the supplementary group list.

NAME

gethostbyaddr, gethostbyname — network host database functions

SYNOPSIS

```
#include <netdb.h>
```

OB

```
struct hostent *gethostbyaddr(const void *addr, socklen_t len,
       int type);
struct hostent *gethostbyname(const char *name);
```

DESCRIPTION

These functions shall retrieve information about hosts. This information is considered to be stored in a database that can be accessed sequentially or randomly. Implementation of this database is unspecified.

Note: In many cases it is implemented by the Domain Name System, as documented in RFC 1034, RFC 1035, and RFC 1886.

Entries shall be returned in **hostent** structures.

The *gethostbyaddr*( ) function shall return an entry containing addresses of address family *type* for the host with address *addr*. The *len* argument contains the length of the address pointed to by *addr*. The *gethostbyaddr*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The *gethostbyname*( ) function shall return an entry containing addresses of address family AF_INET for the host with name *name*. The *gethostbyname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The *addr* argument of *gethostbyaddr*( ) shall be an **in_addr** structure when *type* is AF_INET. It contains a binary format (that is, not null-terminated) address in network byte order. The *gethostbyaddr*( ) function is not guaranteed to return addresses of address families other than AF_INET, even when such addresses exist in the database.

If *gethostbyaddr*( ) returns successfully, then the *h_addrtype* field in the result shall be the same as the *type* argument that was passed to the function, and the *h_addr_list* field shall list a single address that is a copy of the *addr* argument that was passed to the function.

The *name* argument of *gethostbyname*( ) shall be a node name; the behavior of *gethostbyname*( ) when passed a numeric address string is unspecified. For IPv4, a numeric address string shall be in the dotted-decimal notation described in *inet_addr*( ).

If *name* is not a numeric address string and is an alias for a valid host name, then *gethostbyname*( ) shall return information about the host name to which the alias refers, and *name* shall be included in the list of aliases returned.

RETURN VALUE

Upon successful completion, these functions shall return a pointer to a **hostent** structure if the requested entry was found, and a null pointer if the end of the database was reached or the requested entry was not found.

Upon unsuccessful completion, *gethostbyaddr*( ) and *gethostbyname*( ) shall set *h_errno* to indicate the error.

ERRORS

These functions shall fail in the following cases. The *gethostbyaddr*( ) and *gethostbyname*( ) functions shall set *h_errno* to the value shown in the list below. Any changes to *errno* are unspecified.

[HOST_NOT_FOUND]	No such host is known.
[NO_DATA]	The server recognized the request and the name, but no address is available. Another type of request to the name server for the domain might return an answer.
[NO_RECOVERY]	An unexpected server failure occurred which cannot be recovered.
[TRY_AGAIN]	A temporary and possibly transient error occurred, such as a failure of a server to respond.

EXAMPLES

None.

APPLICATION USAGE

The *gethostbyaddr*( ) and *gethostbyname*( ) functions may return pointers to static data, which may be overwritten by subsequent calls to any of these functions.

The *getaddrinfo*( ) and *getnameinfo*( ) functions are preferred over the *gethostbyaddr*( ) and *gethostbyname*( ) functions.

RATIONALE

None.

FUTURE DIRECTIONS

The *gethostbyaddr*( ) and *gethostbyname*( ) functions may be withdrawn in a future version.

SEE ALSO

endhostent( ), *endservent*( ), *gai_strerror*( ), *getaddrinfo*( ), *h_errno*, *inet_addr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

gethostbyname — network host database functions

SYNOPSIS

```
#include <netdb.h>
```

OB
```
struct hostent *gethostbyname(const char *name);
```

DESCRIPTION

Refer to *gethostbyaddr*( ).

NAME

gethostent — network host database functions

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostent(void);
```

DESCRIPTION

Refer to *endhostent*( ).

NAME

gethostid — get an identifier for the current host

SYNOPSIS

XSI

```
#include <unistd.h>

long gethostid(void);
```

DESCRIPTION

The *gethostid*( ) function shall retrieve a 32-bit identifier for the current host.

RETURN VALUE

Upon successful completion, *gethostid*( ) shall return an identifier for the current host.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This volume of IEEE Std 1003.1-200x does not define the domain in which the return value is unique.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

random( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

gethostname — get name of current host

SYNOPSIS

```
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

DESCRIPTION

The *gethostname*( ) function shall return the standard host name for the current machine. The *namelen* argument shall specify the size of the array pointed to by the *name* argument. The returned name shall be null-terminated, except that if *namelen* is an insufficient length to hold the host name, then the returned name shall be truncated and it is unspecified whether the returned name is null-terminated.

Host names are limited to {HOST_NAME_MAX} bytes.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, –1 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

gethostid( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The Open Group Base Resolution bwg2001-008 is applied, changing the *namelen* parameter from **socklen_t** to **size_t**.

NAME

getitimer, setitimer — get and set value of interval timer

SYNOPSIS

XSI

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
    struct itimerval *restrict ovalue);

```

DESCRIPTION

The *getitimer*( ) function shall store the current value of the timer specified by *which* into the structure pointed to by *value*. The *setitimer*( ) function shall set the timer specified by *which* to the value specified in the structure pointed to by *value*, and if *ovalue* is not a null pointer, stores the previous value of the timer in the structure pointed to by *ovalue*.

A timer value is defined by the **itimerval** structure, specified in **<sys/time.h>**. If *it_value* is non-zero, it shall indicate the time to the next timer expiration. If *it_interval* is non-zero, it shall specify a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 shall disable a timer, regardless of the value of *it_interval*. Setting *it_interval* to 0 shall disable a timer after its next expiration (assuming *it_value* is non-zero).

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value.

An XSI-conforming implementation provides each process with at least three interval timers, which are indicated by the *which* argument:

ITIMER_REAL	Decrements in real time. A SIGALRM signal is delivered when this timer expires.
ITIMER_VIRTUAL	Decrements in process virtual time. It runs only when the process is executing. A SIGVTALRM signal is delivered when it expires.
ITIMER_PROF	Decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the ITIMER_PROF timer expires, the SIGPROF signal is delivered.

The interaction between *setitimer*( ) and any of *alarm*( ), *sleep*( ), or *usleep*( ) is unspecified.

RETURN VALUE

Upon successful completion, *getitimer*( ) or *setitimer*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setitimer*( ) function shall fail if:

[EINVAL]	The *value* argument is not in canonical form. (In canonical form, the number of microseconds is a non-negative integer less than 1,000,000 and the number of seconds is a non-negative integer.)

The *getitimer*( ) and *setitimer*( ) functions may fail if:

[EINVAL]	The *which* argument is not recognized.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *sleep*( ), *timer_getoverrun*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X / OPEN UNIX extension to BASE.

Issue 6

The **restrict** keyword is added to the *setitimer*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

getlogin, getlogin_r — get login name

SYNOPSIS

```
#include <unistd.h>

char *getlogin(void);
```

TSF
```
int getlogin_r(char *name, size_t namesize);
```

DESCRIPTION

The *getlogin*( ) function shall return a pointer to a string containing the user name associated by the login activity with the controlling terminal of the current process. If *getlogin*( ) returns a non-null pointer, then that pointer points to the name that the user logged in under, even if there are several login names with the same user ID.

The *getlogin*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *getlogin_r*( ) function shall put the name associated by the login activity with the controlling terminal of the current process in the character array pointed to by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum size of the login name is {LOGIN_NAME_MAX}. |

If *getlogin_r*( ) is successful, *name* points to the name the user used at login, even if there are several login names with the same user ID.

RETURN VALUE

Upon successful completion, *getlogin*( ) shall return a pointer to the login name or a null pointer if the user's login name cannot be found. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

The return value from *getlogin*( ) may point to static data whose content is overwritten by each call.

TSF If successful, the *getlogin_r*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *getlogin*( ) and *getlogin_r*( ) functions may fail if:

[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.
[ENXIO]	The calling process has no controlling terminal.

The *getlogin_r*( ) function may fail if:

TSF [ERANGE]	The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

EXAMPLES

Getting the User Login Name

The following example calls the *getlogin*( ) function to obtain the name of the user associated with the calling process, and passes this information to the *getpwnam*( ) function to get the associated user database information.

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
    }
```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid*(*geteuid*( )) shall return the name associated with the effective user ID of the process; *getlogin*( ) shall return the name associated with the current login activity; and *getpwuid*(*getuid*( )) shall return the name associated with the real user ID of the process.

The *getlogin_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

The *getlogin*( ) function returns a pointer to the user's login name. The same user ID may be shared by several login names. If it is desired to get the user database entry that is used during login, the result of *getlogin*( ) should be used to provide the argument to the *getpwnam*( ) function. (This might be used to determine the user's login shell, particularly where a single user has multiple login shells with distinct login names, but the same user ID.)

The information provided by the *cuserid*( ) function, which was originally defined in the POSIX.1-1988 standard and subsequently removed, can be obtained by the following:

```
getpwuid(geteuid())
```

while the information provided by historical implementations of *cuserid*( ) can be obtained by:

```
getpwuid(getuid())
```

The thread-safe version of this function places the user name in a user-supplied buffer and returns a non-zero value if it fails. The non-thread-safe version may return the name in a static data area that may be overwritten by each call.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwnam( ), *getpwuid*( ), *geteuid*( ), *getuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getlogin_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getlogin*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getlogin_r*( ) function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

NAME

getmsg, getpmsg — receive next message from a STREAMS file (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>

int getmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict flagsp);
int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```

DESCRIPTION

The *getmsg*( ) function shall retrieve the contents of a message located at the head of the STREAM head read queue associated with a STREAMS file and place the contents into one or more buffers. The message contains either a data part, a control part, or both. The data and control parts of the message shall be placed into separate buffers, as described below. The semantics of each part are defined by the originator of the message.

The *getpmsg*( ) function shall be equivalent to *getmsg*( ), except that it provides finer control over the priority of the messages received. Except where noted, all requirements on *getmsg*( ) also pertain to *getpmsg*( ).

The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the *buf* member points to a buffer in which the data or control information is to be placed, and the *maxlen* member indicates the maximum number of bytes this buffer can hold. On return, the *len* member shall contain the number of bytes of data or control information actually received. The *len* member shall be set to 0 if there is a zero-length control or data part and *len* shall be set to −1 if no data or control information is present in the message.

When *getmsg*( ) is called, *flagsp* should point to an integer that indicates the type of message the process is able to receive. This is described further below.

The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the *maxlen* member is −1, the control (or data) part of the message shall not be processed and shall be left on the STREAM head read queue, and if the *ctlptr* (or *dataptr*) is not a null pointer, *len* shall be set to −1. If the *maxlen* member is set to 0 and there is a zero-length control (or data) part, that zero-length part shall be removed from the read queue and *len* shall be set to 0. If the *maxlen* member is set to 0 and there are more than 0 bytes of control (or data) information, that information shall be left on the read queue and *len* shall be set to 0. If the *maxlen* member in *ctlptr* (or *dataptr*) is less than the control (or data) part of the message, *maxlen* bytes shall be retrieved. In this case, the remainder of the message shall be left on the STREAM head read queue and a non-zero return value shall be provided.

By default, *getmsg*( ) shall process the first available message on the STREAM head read queue. However, a process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to RS_HIPRI. In this case, *getmsg*( ) shall only process the next message if it is a high-priority message. When the integer pointed to by *flagsp* is 0, any available message shall be retrieved. In this case, on return, the integer pointed to by *flagsp* shall be set to RS_HIPRI if a high-priority message was retrieved, or 0 otherwise.

For *getpmsg*( ), the flags are different. The *flagsp* argument points to a bitmask with the following mutually-exclusive flags defined: MSG_HIPRI, MSG_BAND, and MSG_ANY. Like *getmsg*( ), *getpmsg*( ) shall process the first available message on the STREAM head read queue. A process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to MSG_HIPRI and the integer pointed to by *bandp* to 0. In this case, *getpmsg*( ) shall only process the next message if it is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to MSG_BAND and the integer pointed to by *bandp* to the priority band of interest. In this case, *getpmsg*( ) shall only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a process wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to MSG_ANY and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* shall be set to MSG_HIPRI and the integer pointed to by *bandp* shall be set to 0. Otherwise, the integer pointed to by *flagsp* shall be set to MSG_BAND and the integer pointed to by *bandp* shall be set to the priority band of the message.

If O_NONBLOCK is not set, *getmsg*( ) and *getpmsg*( ) shall block until a message of the type specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is set and a message of the specified type is not present at the front of the read queue, *getmsg*( ) and *getpmsg*( ) shall fail and set *errno* to [EAGAIN].

If a hangup occurs on the STREAM from which messages are retrieved, *getmsg*( ) and *getpmsg*( ) shall continue to operate normally, as described above, until the STREAM head read queue is empty. Thereafter, they shall return 0 in the *len* members of *ctlptr* and *dataptr*.

RETURN VALUE

Upon successful completion, *getmsg*( ) and *getpmsg*( ) shall return a non-negative value. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of the bitwise-logical OR of MORECTL and MOREDATA indicates that both types of information remain. Subsequent *getmsg*( ) and *getpmsg*( ) calls shall retrieve the remainder of the message. However, if a message of higher priority has come in on the STREAM head read queue, the next call to *getmsg*( ) or *getpmsg*( ) shall retrieve that higher-priority message before retrieving the remainder of the previous message.

If the high priority control part of the message is consumed, the message shall be placed back on the queue as a normal message of band 0. Subsequent *getmsg*( ) and *getpmsg*( ) calls shall retrieve the remainder of the message. If, however, a priority message arrives or already exists on the STREAM head, the subsequent call to *getmsg*( ) or *getpmsg*( ) shall retrieve the higher-priority message before retrieving the remainder of the message that was put back.

Upon failure, *getmsg*( ) and *getpmsg*( ) shall return −1 and set *errno* to indicate the error.

ERRORS

The *getmsg*( ) and *getpmsg*( ) functions shall fail if:

[EAGAIN]	The O_NONBLOCK flag is set and no messages are available.
[EBADF]	The *fildes* argument is not a valid file descriptor open for reading.
[EBADMSG]	The queued message to be read is not valid for *getmsg*( ) or *getpmsg*( ) or a pending file descriptor is at the STREAM head.
[EINTR]	A signal was caught during *getmsg*( ) or *getpmsg*( ).
[EINVAL]	An illegal value was specified by *flagsp*, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

[ENOSTR] A STREAM is not associated with *fildes*.

In addition, *getmsg*( ) and *getpmsg*( ) shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *getmsg*( ) or *getpmsg*( ) but reflects the prior error.

EXAMPLES

Getting Any Message

In the following example, the value of *fd* is assumed to refer to an open STREAMS file. The call to *getmsg*( ) retrieves any available message on the associated STREAM-head read queue, returning control and data information to the buffers pointed to by *ctrlbuf* and *databuf*, respectively.

```
#include <stropts.h>
...
int fd;
char ctrlbuf[128];
char databuf[512];
struct strbuf ctrl;
struct strbuf data;
int flags = 0;
int ret;

ctrl.buf = ctrlbuf;
ctrl.maxlen = sizeof(ctrlbuf);

data.buf = databuf;
data.maxlen = sizeof(databuf);

ret = getmsg (fd, &ctrl, &data, &flags);
```

Getting the First Message off the Queue

In the following example, the call to *getpmsg*( ) retrieves the first available message on the associated STREAM-head read queue.

```
#include <stropts.h>
...

int fd;
char ctrlbuf[128];
char databuf[512];
struct strbuf ctrl;
struct strbuf data;
int band = 0;
int flags = MSG_ANY;
int ret;

ctrl.buf = ctrlbuf;
ctrl.maxlen = sizeof(ctrlbuf);

data.buf = databuf;
data.maxlen = sizeof(databuf);

ret = getpmsg (fd, &ctrl, &data, &band, &flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

poll( ), *putmsg*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

A paragraph regarding ‘‘high-priority control parts of messages’’ is added to the RETURN VALUE section.

Issue 6

This function is marked as part of the XSI STREAMS Option Group.

The **restrict** keyword is added to the *getmsg*( ) and *getpmsg*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

NAME

getnameinfo — get name information

SYNOPSIS

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *restrict sa, socklen_t salen,
    char *restrict node, socklen_t nodelen, char *restrict service,
    socklen_t servicelen, unsigned flags);
```

DESCRIPTION

The *getnameinfo*( ) function shall translate a socket address to a node name and service location, all of which are defined as in *getaddrinfo*( ).

The *sa* argument points to a socket address structure to be translated.

If the *node* argument is non-NULL and the *nodelen* argument is non-zero, then the *node* argument points to a buffer able to contain up to *nodelen* characters that receives the node name as a null-terminated string. If the *node* argument is NULL or the *nodelen* argument is zero, the node name shall not be returned. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name.

If the *service* argument is non-NULL and the *servicelen* argument is non-zero, then the *service* argument points to a buffer able to contain up to *servicelen* bytes that receives the service name as a null-terminated string. If the *service* argument is NULL or the *servicelen* argument is zero, the service name shall not be returned. If the service's name cannot be located, the numeric form of the service address (for example, its port number) shall be returned instead of its name.

The *flags* argument is a flag that changes the default actions of the function. By default the fully-qualified domain name (FQDN) for the host shall be returned, but:

- If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN shall be returned for local hosts.
- If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address shall be returned instead of its name, under all circumstances.
- If the flag bit NI_NAMEREQD is set, an error shall be returned if the host's name cannot be located.
- If the flag bit NI_NUMERICSERV is set, the numeric form of the service address shall be returned (for example, its port number) instead of its name, under all circumstances.
- If the flag bit NI_DGRAM is set, this indicates that the service is a datagram service (SOCK_DGRAM). The default behavior shall assume that the service is a stream service (SOCK_STREAM).

Notes:

1. The two NI_NUMERICxxx flags are required to support the **−n** flag that many commands provide.
2. The NI_DGRAM flag is required for the few AF_INET and AF_INET6 port numbers (for example, [512,514]) that represent different services for UDP and TCP.

The *getnameinfo*( ) function shall be thread-safe.

RETURN VALUE

A zero return value for *getnameinfo*( ) indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed in the ERRORS section.

Upon successful completion, *getnameinfo*( ) shall return the *node* and *service* names, if requested, in the buffers provided. The returned names are always null-terminated strings.

ERRORS

The *getnameinfo*( ) function shall fail and return the corresponding value if:

[EAI_AGAIN]	The name could not be resolved at this time. Future attempts may succeed.
[EAI_BADFLAGS]	The *flags* had an invalid value.
[EAI_FAIL]	A non-recoverable error occurred.
[EAI_FAMILY]	The address family was not recognized or the address length was invalid for the specified family.
[EAI_MEMORY]	There was a memory allocation failure.
[EAI_NONAME]	The name does not resolve for the supplied parameters. NI_NAMEREQD is set and the host's name cannot be located, or both *nodename* and *servname* were null.
[EAI_SYSTEM]	A system error occurred. The error code can be found in *errno*.

EXAMPLES

None.

APPLICATION USAGE

If the returned values are to be used as part of any further name resolution (for example, passed to *getaddrinfo*( )), applications should provide buffers large enough to store any result possible on the system.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

gai_strerror( ), *getaddrinfo*( ), *getservbyname*( ), *getservbyport*( ), *inet_ntop*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getnameinfo*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

getnetbyaddr — network database functions

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetbyaddr(uint32_t net, int type);
```

DESCRIPTION

Refer to *endnetent*( ).

NAME

getnetbyname — network database functions

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetbyname(const char *name);
```

DESCRIPTION

Refer to *endnetent*( ).

NAME

getnetent — network database functions

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent(void);
```

DESCRIPTION

Refer to *endnetent*( ).

NAME

getopt, optarg, opterr, optind, optopt — command option parsing

SYNOPSIS

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

DESCRIPTION

The *getopt*( ) function is a command-line parser that shall follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9, and 10 in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The parameters *argc* and *argv* are the argument count and argument array as passed to *main*( ) (see *exec*). The argument *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument. All option characters allowed by Utility Syntax Guideline 3 are allowed in *optstring*. The implementation may accept other characters as an extension.

The variable *optind* is the index of the next element of the *argv*[] vector to be processed. It shall be initialized to 1 by the system, and *getopt*( ) shall update it when it finishes with each element of *argv*[ ]. When an element of *argv*[ ] contains multiple option characters, it is unspecified how *getopt*( ) determines which options have already been processed.

The *getopt*( ) function shall return the next option character (if one is found) from *argv* that matches a character in *optstring*, if there is one that matches. If the option takes an argument, *getopt*( ) shall set the variable *optarg* to point to the option-argument as follows:

1. If the option was the last character in the string pointed to by an element of *argv*, then *optarg* shall contain the next element of *argv*, and *optind* shall be incremented by 2. If the resulting value of *optind* is greater than *argc*, this indicates a missing option-argument, and *getopt*( ) shall return an error indication.
2. Otherwise, *optarg* shall point to the string following the option character in that element of *argv*, and *optind* shall be incremented by 1.

If, when *getopt*( ) is called:

`argv[optind]`	is a null pointer
`*argv[optind]`	is not the character `–`
`argv[optind]`	points to the string `"–"`

getopt( ) shall return –1 without changing *optind*. If:

`argv[optind]`	points to the string `"––"`

getopt( ) shall return –1 after incrementing *optind*.

If *getopt*( ) encounters an option character that is not contained in *optstring*, it shall return the question-mark (`'?'`) character. If it detects a missing option-argument, it shall return the colon character (`':'`) if the first character of *optstring* was a colon, or a question-mark character (`'?'`) otherwise. In either case, *getopt*( ) shall set the variable *optopt* to the option character that caused the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring* is not a colon, *getopt*( ) shall also print a diagnostic message to *stderr* in the format specified for the *getopts* utility.

The *getopt*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *getopt*( ) function shall return the next option character specified on the command line.

A colon (’:’) shall be returned if *getopt*( ) detects a missing argument and the first character of *optstring* was a colon (’:’).

A question mark (’?’) shall be returned if *getopt*( ) encounters an option character not in *optstring* or detects a missing argument and the first character of *optstring* was not a colon (’:’).

Otherwise, *getopt*( ) shall return −1 when all command line options are parsed.

ERRORS

No errors are defined.

EXAMPLES

Parsing Command Line Options

The following code fragment shows how you might process the arguments for a utility that can take the mutually-exclusive options *a* and *b* and the options *f* and *o*, both of which require arguments:

```
#include <unistd.h>

int
main(int argc, char *argv[ ])
{
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind, optopt;
    . . .
    while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
        switch(c) {
        case ’a’:
            if (bflg)
                errflg++;
            else
                aflg++;
            break;
        case ’b’:
            if (aflg)
                errflg++;
            else {
                bflg++;
                bproc();
            }
            break;
        case ’f’:
            ifile = optarg;
            break;
        case ’o’:
            ofile = optarg;
            break;
```

```
            case ’:’:       /* -f or -o without operand */
                    fprintf(stderr,
                            "Option -%c requires an operand\n", optopt);
                    errflg++;
                    break;
        case ’?’:
                    fprintf(stderr,
                            "Unrecognized option: -%c\n", optopt);
            errflg++;
        }
    }
    if (errflg) {
        fprintf(stderr, "usage: . . . ");
        exit(2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], R_OK)) {
    . . .
}
```

This code accepts any of the following as equivalent:

```
cmd −ao arg path path
cmd −a −o arg path path
cmd −o arg −a path path
cmd −a −o arg −− path path
cmd −a −oarg path path
cmd −aoarg path path
```

Checking Options and Arguments

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```
#include <unistd.h>
#include <stdio.h>
...
int c;
char *filename;
extern char *optarg;
extern int optind, optopt, opterr;
...
while ((c = getopt(argc, argv, ":abf:")) != -1) {
    switch(c) {
    case ’a’:
        printf("a is set\n");
        break;
    case ’b’:
        printf("b is set\n");
        break;
    case ’f’:
        filename = optarg;
        printf("filename is %s\n", filename);
        break;
```

```
        case ':':
            printf("-%c without filename\n", optopt);
            break;
        case '?':
            printf("unknown arg %c\n", optopt);
            break;
        }
    }
```

Selecting Options from the Command Line

The following example selects the type of database routines the user wants to use based on the *Options* argument.

```
#include <unistd.h>
#include <string.h>
...
char *Options = "hdbtl";
...
int dbtype, i;
char c;
char *st;
...
dbtype = 0;
while ((c = getopt(argc, argv, Options)) != −1) {
    if ((st = strchr(Options, c)) != NULL) {
        dbtype = st - Options;
        break;
    }
}
```

APPLICATION USAGE

The *getopt*( ) function is only required to support option characters included in Utility Syntax Guideline 3. Many historical implementations of *getopt*( ) support other characters as options. This is an allowed extension, but applications that use extensions are not maximally portable. Note that support for multi-byte option characters is only possible when such characters can be represented as type **int**.

RATIONALE

The *optopt* variable represents historical practice and allows the application to obtain the identity of the invalid option.

The description has been written to make it clear that *getopt*( ), like the *getopts* utility, deals with option-arguments whether separated from the option by <blank>s or not. Note that the requirements on *getopt*( ) and *getopts* are more stringent than the Utility Syntax Guidelines.

The *getopt*( ) function shall return −1, rather than EOF, so that **<stdio.h>** is not required.

The special significance of a colon as the first character of *optstring* makes *getopt*( ) consistent with the *getopts* utility. It allows an application to make a distinction between a missing argument and an incorrect option letter without having to examine the option letter. It is true that a missing argument can only be detected in one case, but that is a case that has to be considered.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *getopt*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

IEEE PASC Interpretation 1003.2 #150 is applied.

NAME

getpeername — get the name of the peer socket

SYNOPSIS

```
#include <sys/socket.h>

int getpeername(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

DESCRIPTION

The *getpeername*( ) function shall retrieve the peer address of the specified socket, store this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by *address* is unspecified.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getpeername*( ) function shall fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[EINVAL]	The socket has been shut down.
[ENOTCONN]	The socket is not connected or otherwise has not had the peer prespecified.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The operation is not supported for the socket protocol.

The *getpeername*( ) function may fail if:

[ENOBUFS]	Insufficient resources were available in the system to complete the call.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *getsockname*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getpeername*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

getpgid — get the process group ID for a process

SYNOPSIS

XSI
```
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

DESCRIPTION

The *getpgid*( ) function shall return the process group ID of the process whose process ID is equal to *pid*. If *pid* is equal to 0, *getpgid*( ) shall return the process group ID of the calling process.

RETURN VALUE

Upon successful completion, *getpgid*( ) shall return a process group ID. Otherwise, it shall return (**pid_t**)−1 and set *errno* to indicate the error.

ERRORS

The *getpgid*( ) function shall fail if:

[EPERM]	The process whose process ID is equal to *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of that process from the calling process.
[ESRCH]	There is no process with a process ID equal to *pid*.

The *getpgid*( ) function may fail if:

[EINVAL]	The value of the *pid* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpgrp*( ), *getpid*( ), *getsid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

getpgrp — get the process group ID of the calling process

SYNOPSIS

```
#include <unistd.h>

pid_t getpgrp(void);
```

DESCRIPTION

The *getpgrp*( ) function shall return the process group ID of the calling process.

RETURN VALUE

The *getpgrp*( ) function shall always be successful and no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

4.3 BSD provides a *getpgrp*( ) function that returns the process group ID for a specified process. Although this function supports job control, all known job control shells always specify the calling process with this function. Thus, the simpler System V *getpgrp*( ) suffices, and the added complexity of the 4.3 BSD *getpgrp*( ) is provided by the XSI extension *getpgid*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpgid*( ), *getpid*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getpid — get the process ID

SYNOPSIS

```
#include <unistd.h>

pid_t getpid(void);
```

DESCRIPTION

The *getpid*( ) function shall return the process ID of the calling process.

RETURN VALUE

The *getpid*( ) function shall always be successful and no return value is reserved to indicate an error. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpgrp*( ), *getppid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getpmsg — receive next message from a STREAMS file

SYNOPSIS

XSI

```
#include <stropts.h>

int getpmsg(int fildes, struct strbuf *restrict ctlptr,
    struct strbuf *restrict dataptr, int *restrict bandp,
    int *restrict flagsp);
```

DESCRIPTION

Refer to *getmsg*( ).

NAME

getppid — get the parent process ID

SYNOPSIS

```
#include <unistd.h>

pid_t getppid(void);
```

DESCRIPTION

The *getppid*( ) function shall return the parent process ID of the calling process.

RETURN VALUE

The *getppid*( ) function shall always be successful and no return value is reserved to indicate an error. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpgid*( ), *getpgrp*( ), *getpid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getpriority, setpriority — get and set the nice value

SYNOPSIS

XSI

```
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```

DESCRIPTION

The *getpriority*( ) function shall obtain the nice value of a process, process group, or user. The *setpriority*( ) function shall set the nice value of a process, process group, or user to *value*+{NZERO}.

Target processes are specified by the values of the *which* and *who* arguments. The *which* argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or an effective user ID, respectively. A 0 value for the *who* argument specifies the current process, process group, or user.

The nice value set with *setpriority*( ) shall be applied to the process. If the process is multi-threaded, the nice value shall affect all system scope threads in the process.

If more than one process is specified, *getpriority*( ) shall return value {NZERO} less than the lowest nice value pertaining to any of the specified processes, and *setpriority*( ) shall set the nice values of all of the specified processes to *value*+{NZERO}.

The default nice value is {NZERO}; lower nice values shall cause more favorable scheduling. While the range of valid nice values is [0,{NZERO}*2–1], implementations may enforce more restrictive limits. If *value*+{NZERO} is less than the system's lowest supported nice value, *setpriority*( ) shalls et the nice value to the lowest supported value; if *value*+{NZERO} is greater than the system's highest supported nice value, *setpriority*( ) shall set the nice value to the highest supported value.

Only a process with appropriate privileges can lower its nice value.

PS|TPS Any processes or threads using SCHED_FIFO or SCHED_RR shall be unaffected by a call to *setpriority*( ). This is not considered an error. A process which subsequently reverts to SCHED_OTHER need not have its priority affected by such a *setpriority*( ) call.

The effect of changing the nice value may vary depending on the process-scheduling algorithm in effect.

Since *getpriority*( ) can return the value –1 on successful completion, it is necessary to set *errno* to 0 prior to a call to *getpriority*( ). If *getpriority*( ) returns the value –1, then *errno* can be checked to see if an error occurred or if the value is a legitimate nice value.

RETURN VALUE

Upon successful completion, *getpriority*( ) shall return an integer in the range from –{NZERO} to {NZERO}–1. Otherwise, –1 shall be returned and *errno* set to indicate the error.

Upon successful completion, *setpriority*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getpriority*( ) and *setpriority*( ) functions shall fail if:

[ESRCH]	No process could be located using the *which* and *who* argument values specified.

[EINVAL]	The value of the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID, or user ID.

In addition, *setpriority*( ) may fail if:

[EPERM]	A process was located, but neither the real nor effective user ID of the executing process match the effective user ID of the process whose nice value is being changed.
[EACCES]	A request was made to change the nice value to a lower numeric value and the current process does not have appropriate privileges.

EXAMPLES

Using getpriority( )

The following example returns the current scheduling priority for the process ID returned by the call to *getpid*( ).

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int ret;

pid = getpid();
ret = getpriority(which, pid);
```

Using setpriority( )

The following example sets the priority for the current process ID to –20.

```
#include <sys/resource.h>
...
int which = PRIO_PROCESS;
id_t pid;
int priority = -20;
int ret;

pid = getpid();
ret = setpriority(which, pid, priority);
```

APPLICATION USAGE

The *getpriority*( ) and *setpriority*( ) functions work with an offset nice value (nice value –{NZERO}). The nice value is in the range [0,2*{NZERO} –1], while the return value for *getpriority*( ) and the third parameter for *setpriority*( ) are in the range [–{NZERO},{NZERO} –1].

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

nice( ), *sched_get_priority_max*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/resource.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is reworded in terms of the nice value rather than *priority* to avoid confusion with functionality in the POSIX Realtime Extension.

NAME

getprotobyname — network protocol database functions

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotobyname(const char *name);
```

DESCRIPTION

Refer to *endprotoent*( ).

NAME

getprotobynumber — network protocol database functions

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotobynumber(int proto);
```

DESCRIPTION

Refer to *endprotoent*( ).

NAME

getprotoent — network protocol database functions

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent(void);
```

DESCRIPTION

Refer to *endprotoent*( ).

NAME

getpwent — get user database entry

SYNOPSIS

XSI
```
#include <pwd.h>

struct passwd *getpwent(void);
```

DESCRIPTION

Refer to *endpwent*( ).

NAME

getpwnam, getpwnam_r — search user database for a name

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwnam(const char *name);
int getpwnam_r(const char *name, struct passwd *pwd, char *buffer,
    size_t bufsize, struct passwd **result);
```

TSF

DESCRIPTION

The *getpwnam*( ) function shall search the user database for an entry with a matching *name*.

The *getpwnam*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

Applications wishing to check for error situations should set *errno* to 0 before calling *getpwnam*( ). If *getpwnam*( ) returns a null pointer and *errno* is non-zero, an error occurred.

TSF The *getpwnam_r*( ) function shall update the **passwd** structure pointed to by *pwd* and store a pointer to that structure at the location pointed to by *result*. The structure shall contain an entry from the user database with a matching *name*. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX} *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on error or if the requested entry is not found.

RETURN VALUE

The *getpwnam*( ) function shall return a pointer to a **struct passwd** with the structure as defined in **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

TSF If successful, the *getpwnam_r*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *getpwnam*( ) and *getpwnam_r*( ) functions may fail if:

[EIO]	An I/O error has occurred.
[EINTR]	A signal was caught during *getpwnam*( ).
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

The *getpwnam_r*( ) function may fail if:

TSF [ERANGE]	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **passwd** structure.

EXAMPLES

Getting an Entry for the Login Name

The following example uses the *getlogin*( ) function to return the name of the user who logged in; this information is passed to the *getpwnam*( ) function to get the user database entry for that user.

```
#include <sys/types.h>
#include <pwd.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
...
char *lgn;
struct passwd *pw;
...
if ((lgn = getlogin()) == NULL || (pw = getpwnam(lgn)) == NULL) {
    fprintf(stderr, "Get of user information failed.\n"); exit(1);
}
...
```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid*(*geteuid*( )) returns the name associated with the effective user ID of the process; *getlogin*( ) returns the name associated with the current login activity; and *getpwuid*(*getuid*( )) returns the name associated with the real user ID of the process.

The *getpwnam_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwuid( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pwd.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getpwnam_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getpwnam*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getpwnam_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *name*.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EMFILE], [ENFILE], and [ENXIO] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

NAME

getpwuid, getpwuid_r — search user database for a user ID

SYNOPSIS

```
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

TSF

```
int getpwuid_r(uid_t uid, struct passwd *pwd, char *buffer,
    size_t bufsize, struct passwd **result);
```

DESCRIPTION

The *getpwuid*( ) function shall search the user database for an entry with a matching *uid*.

The *getpwuid*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

Applications wishing to check for error situations should set *errno* to 0 before calling *getpwuid*( ). If *getpwuid*( ) returns a null pointer and *errno* is set to non-zero, an error occurred.

TSF The *getpwuid_r*( ) function shall update the **passwd** structure pointed to by *pwd* and store a pointer to that structure at the location pointed to by *result*. The structure shall contain an entry from the user database with a matching *uid*. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. The maximum size needed for this buffer can be determined with the {_SC_GETPW_R_SIZE_MAX} *sysconf*( ) parameter. A NULL pointer shall be returned at the location pointed to by *result* on error or if the requested entry is not found. |

RETURN VALUE

The *getpwuid*( ) function shall return a pointer to a **struct passwd** with the structure as defined in **<pwd.h>** with a matching entry if found. A null pointer shall be returned if the requested entry is not found, or an error occurs. On error, *errno* shall be set to indicate the error.

The return value may point to a static area which is overwritten by a subsequent call to *getpwent*( ), *getpwnam*( ), or *getpwuid*( ).

TSF If successful, the *getpwuid_r*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *getpwuid*( ) and *getpwuid_r*( ) functions may fail if:

[EIO]	An I/O error has occurred.
[EINTR]	A signal was caught during *getpwuid*( ).
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

The *getpwuid_r*( ) function may fail if:

[ERANGE]	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **passwd** structure.

TSF

EXAMPLES

Getting an Entry for the Root User

The following example gets the user database entry for the user with user ID 0 (root).

```
#include <sys/types.h>
#include <pwd.h>
...
uid_t id = 0;
struct passwd *pwd;

pwd = getpwuid(id);
```

Finding the Name for the Effective User ID

The following example defines *pws* as a pointer to a structure of type **passwd**, which is used to store the structure pointer returned by the call to the *getpwuid*( ) function. The *geteuid*( ) function shall return the effective user ID of the calling process; this is used as the search criteria for the *getpwuid*( ) function. The call to *getpwuid*( ) shall return a pointer to the structure containing that user ID value.

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
...
struct passwd *pws;
pws = getpwuid(geteuid());
```

Finding an Entry in the User Database

The following example uses *getpwuid*( ) to search the user database for a user ID that was previously stored in a **stat** structure, then prints out the user name if it is found. If the user is not found, the program prints the numeric value of the user ID for the entry.

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
...
struct stat statbuf;
struct passwd *pwd;
...
if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8d", statbuf.st_uid);
```

APPLICATION USAGE

Three names associated with the current process can be determined: *getpwuid*(*geteuid*( )) returns the name associated with the effective user ID of the process; *getlogin*( ) returns the name associated with the current login activity; and *getpwuid*(*getuid*( )) returns the name associated with the real user ID of the process.

The *getpwuid_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpwnam( ), *geteuid*( ), *getuid*( ), *getlogin*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pwd.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from System V Release 2.0.

Issue 5

Normative text previously in the APPLICATION USAGE section is moved to the RETURN VALUE section.

The *getpwuid_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *getpwuid*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *getpwuid_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U028/3 is applied, correcting text in the DESCRIPTION describing matching the *uid*.

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EIO], [EINTR], [EMFILE], and [ENFILE] optional error conditions are added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

IEEE PASC Interpretation 1003.1 #116 is applied, changing the description of the size of the buffer from *bufsize* characters to bytes.

NAME

getrlimit, setrlimit — control maximum resource consumption

SYNOPSIS

XSI

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);

```

DESCRIPTION

The *getrlimit*( ) function shall get, and the *setrlimit*( ) function shall set, limits on the consumption of a variety of resources.

Each call to either *getrlimit*( ) or *setrlimit*( ) identifies a specific resource to be operated upon as well as a resource limit. A resource limit is represented by an **rlimit** structure. The *rlim_cur* member specifies the current or soft limit and the *rlim_max* member specifies the maximum or hard limit. Soft limits may be changed by a process to any value that is less than or equal to the hard limit. A process may (irreversibly) lower its hard limit to any value that is greater than or equal to the soft limit. Only a process with appropriate privileges can raise a hard limit. Both hard and soft limits can be changed in a single call to *setrlimit*( ) subject to the constraints described above.

The value RLIM_INFINITY, defined in **<sys/resource.h>**, shall be considered to be larger than any other limit value. If a call to *getrlimit*( ) returns RLIM_INFINITY for a resource, it means the implementation shall not enforce limits on that resource. Specifying RLIM_INFINITY as any resource limit value on a successful call to *setrlimit*( ) shall inhibit enforcement of that resource limit.

The following resources are defined:

RLIMIT_CORE	This is the maximum size of a core file, in bytes, that may be created by a process. A limit of 0 shall prevent the creation of a core file. If this limit is exceeded, the writing of a core file shall terminate at this size.
RLIMIT_CPU	This is the maximum amount of CPU time, in seconds, used by a process. If this limit is exceeded, SIGXCPU shall be generated for the process. If the process is catching or ignoring SIGXCPU, or all threads belonging to that process are blocking SIGXCPU, the behavior is unspecified.
RLIMIT_DATA	This is the maximum size of a process' data segment, in bytes. If this limit is exceeded, the *malloc*( ) function shall fail with *errno* set to [ENOMEM].
RLIMIT_FSIZE	This is the maximum size of a file, in bytes, that may be created by a process. If a write or truncate operation would cause this limit to be exceeded, SIGXFSZ shall be generated for the thread. If the thread is blocking, or the process is catching or ignoring SIGXFSZ, continued attempts to increase the size of a file from end-of-file to beyond the limit shall fail with *errno* set to [EFBIG].
RLIMIT_NOFILE	This is a number one greater than the maximum value that the system may assign to a newly-created descriptor. If this limit is exceeded, functions that allocate new file descriptors may fail with *errno* set to [EMFILE]. This limit constrains the number of file descriptors that a process may allocate.
RLIMIT_STACK	This is the maximum size of a process' stack, in bytes. The implementation does not automatically grow the stack beyond this limit.

	If this limit is exceeded, SIGSEGV shall be generated for the thread. If the thread is blocking SIGSEGV, or the process is ignoring or catching SIGSEGV and has not made arrangements to use an alternate stack, the disposition of SIGSEGV shall be set to SIG_DFL before it is generated.
RLIMIT_AS	This is the maximum size of a process' total available memory, in bytes. If this limit is exceeded, the *malloc*( ) and *mmap*( ) functions shall fail with *errno* set to [ENOMEM]. In addition, the automatic stack growth fails with the effects outlined above.

When using the *getrlimit*( ) function, if a resource limit can be represented correctly in an object of type **rlim_t**, then its representation is returned; otherwise, if the value of the resource limit is equal to that of the corresponding saved hard limit, the value returned shall be RLIM_SAVED_MAX; otherwise, the value returned shall be RLIM_SAVED_CUR.

When using the *setrlimit*( ) function, if the requested new limit is RLIM_INFINITY, the new limit shall be ‘‘no limit’’; otherwise, if the requested new limit is RLIM_SAVED_MAX, the new limit shall be the corresponding saved hard limit; otherwise, if the requested new limit is RLIM_SAVED_CUR, the new limit shall be the corresponding saved soft limit; otherwise, the new limit shall be the requested value. In addition, if the corresponding saved limit can be represented correctly in an object of type **rlim_t** then it shall be overwritten with the new limit.

The result of setting a limit to RLIM_SAVED_MAX or RLIM_SAVED_CUR is unspecified unless a previous call to *getrlimit*( ) returned that value as the soft or hard limit for the corresponding resource limit.

The determination of whether a limit can be correctly represented in an object of type **rlim_t** is implementation-defined. For example, some implementations permit a limit whose value is greater than RLIM_INFINITY and others do not.

The *exec* family of functions shall cause resource limits to be saved.

RETURN VALUE

Upon successful completion, *getrlimit*( ) and *setrlimit*( ) shall return 0. Otherwise, these functions shall return −1 and set *errno* to indicate the error.

ERRORS

The *getrlimit*( ) and *setrlimit*( ) functions shall fail if:

[EINVAL]	An invalid *resource* was specified; or in a *setrlimit*( ) call, the new *rlim_cur* exceeds the new *rlim_max*.
[EPERM]	The limit specified to *setrlimit*( ) would have raised the maximum limit value, and the calling process does not have appropriate privileges.

The *setrlimit*( ) function may fail if:

[EINVAL]	The limit specified cannot be lowered because current usage is already higher than the limit.

EXAMPLES

None.

APPLICATION USAGE

If a process attempts to set the hard limit or soft limit for RLIMIT_NOFILE to less than the value of {_POSIX_OPEN_MAX} from **<limits.h>**, unexpected behavior may occur.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *malloc*( ), *open*( ), *sigaltstack*( ), *sysconf*( ), *ulimit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, **<sys/resource.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE and an APPLICATION USAGE section is added.

Large File Summit extensions are added.

NAME

getrusage — get information about resource utilization

SYNOPSIS

XSI

```
#include <sys/resource.h>

int getrusage(int who, struct rusage *r_usage);
```

DESCRIPTION

The *getrusage*( ) function shall provide measures of the resources used by the current process or its terminated and waited-for child processes. If the value of the *who* argument is RUSAGE_SELF, information shall be returned about resources used by the current process. If the value of the *who* argument is RUSAGE_CHILDREN, information shall be returned about resources used by the terminated and waited-for children of the current process. If the child is never waited for (for example, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to SIG_IGN), the resource information for the child process is discarded and not included in the resource information provided by *getrusage*( ). |

The *r_usage* argument is a pointer to an object of type **struct rusage** in which the returned information is stored.

RETURN VALUE

Upon successful completion, *getrusage*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getrusage*( ) function shall fail if:

[EINVAL] The value of the *who* argument is not valid.

EXAMPLES

Using getrusage( )

The following example returns information about the resources used by the current process.

```
#include <sys/resource.h>
...
int who = RUSAGE_SELF;
struct rusage usage;
int ret;

ret = getrusage(who, &usage);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *sigaction*( ), *time*( ), *times*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/resource.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

gets — get a string from a stdin stream

SYNOPSIS

```
#include <stdio.h>

char *gets(char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *gets*( ) function shall read bytes from the standard input stream, *stdin*, into the array pointed to by *s*, until a newline is read or an end-of-file condition is encountered. Any <newline> shall be discarded and a null byte shall be placed immediately after the last byte read into the array.

CX The *gets*( ) function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field shall be marked for update by the first successful execution of *fgetc*( ), *fgets*( ), *fread*( ), *getc*( ), *getchar*( ), *gets*( ), *fscanf*( ), or *scanf*( ) using *stream* that returns data not supplied by a prior call to *ungetc*( ).

RETURN VALUE

Upon successful completion, *gets*( ) shall return *s*. If the stream is at end-of-file, the end-of-file indicator for the stream shall be set and *gets*( ) shall return a null pointer. If a read error occurs, CX the error indicator for the stream shall be set, *gets*( ) shall return a null pointer and set *errno* to indicate the error.

ERRORS

Refer to *fgetc*( ).

EXAMPLES

None.

APPLICATION USAGE

Reading a line that overflows the array pointed to by *s* results in undefined behavior. The use of *fgets*( ) is recommended.

Since the user cannot specify the length of the buffer passed to *gets*( ), use of this function is discouraged. The length of the string read is unlimited. It is possible to overflow this buffer in such a way as to cause applications to fail, or possible system security violations.

It is recommended that the *fgets*( ) function should be used to read input lines.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feof( ), *ferror*( ), *fgets*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

getservbyname — network services database functions

SYNOPSIS

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
```

DESCRIPTION

Refer to *endservent*( ).

NAME

getservbyport — network services database functions

SYNOPSIS

```
#include <netdb.h>

struct servent *getservbyport(int port, const char *proto);
```

DESCRIPTION

Refer to *endservent*( ).

NAME

getservent — network services database functions

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent(void);
```

DESCRIPTION

Refer to *endservent*( ).

NAME

getsid — get the process group ID of a session leader

SYNOPSIS

XSI

```
#include <unistd.h>

pid_t getsid(pid_t pid);
```

DESCRIPTION

The *getsid*( ) function shall obtain the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is (**pid_t**)0, it specifies the calling process. |

RETURN VALUE

Upon successful completion, *getsid*( ) shall return the process group ID of the session leader of the specified process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

ERRORS

The *getsid*( ) function shall fail if:

[EPERM]	The process specified by *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of the session leader of that process from the calling process.
[ESRCH]	There is no process with a process ID equal to *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpid*( ), *getpgid*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

getsockname — get the socket name

SYNOPSIS

```
#include <sys/socket.h>

int getsockname(int socket, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

DESCRIPTION

The *getsockname*( ) function shall retrieve the locally-bound name of the specified socket, store this address in the **sockaddr** structure pointed to by the *address* argument, and store the length of this address in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the socket has not been bound to a local name, the value stored in the object pointed to by *address* is unspecified.

RETURN VALUE

Upon successful completion, 0 shall be returned, the *address* argument shall point to the address of the socket, and the *address_len* argument shall point to the length of the address. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getsockname*( ) function shall fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The operation is not supported for this socket's protocol.

The *getsockname*( ) function may fail if:

[EINVAL]	The socket has been shut down.
[ENOBUFS]	Insufficient resources were available in the system to complete the function.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *getpeername*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getsockname*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

getsockopt — get the socket options

SYNOPSIS

```
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
    void *restrict option_value, socklen_t *restrict option_len);
```

DESCRIPTION

The *getsockopt*( ) function manipulates options associated with a socket.

The *getsockopt*( ) function shall retrieve the value for the option specified by the *option_name* argument for the socket specified by the *socket* argument. If the size of the option value is greater than *option_len*, the value stored in the object pointed to by the *option_value* argument shall be silently truncated. Otherwise, the object pointed to by the *option_len* argument shall be modified to indicate the actual length of the value.

The *level* argument specifies the protocol level at which the option resides. To retrieve options at the socket level, specify the *level* argument as SOL_SOCKET. To retrieve options at other levels, supply the appropriate level identifier for the protocol controlling the option. For example, to indicate that an option is interpreted by the TCP (Transmission Control Protocol), set *level* to IPPROTO_TCP as defined in the **<netinet/in.h>** header.

The socket in use may require the process to have appropriate privileges to use the *getsockopt*( ) function.

The *option_name* argument specifies a single option to be retrieved. It can be one of the following values defined in **<sys/socket.h>**:

SO_DEBUG	Reports whether debugging information is being recorded. This option shall store an **int** value. This is a Boolean option.
SO_ACCEPTCONN	Reports whether socket listening is enabled. This option shall store an **int** value. This is a Boolean option.
SO_BROADCAST	Reports whether transmission of broadcast messages is supported, if this is supported by the protocol. This option shall store an **int** value. This is a Boolean option.
SO_REUSEADDR	Reports whether the rules used in validating addresses supplied to *bind*( ) should allow reuse of local addresses, if this is supported by the protocol. This option shall store an **int** value. This is a Boolean option.
SO_KEEPALIVE	Reports whether connections are kept active with periodic transmission of messages, if this is supported by the protocol. If the connected socket fails to respond to these messages, the connection shall be broken and threads writing to that socket shall be notified with a SIGPIPE signal. This option shall store an **int** value. This is a Boolean option.
SO_LINGER	Reports whether the socket lingers on *close*( ) if data is present. If SO_LINGER is set, the system blocks the process during *close*( ) until it can transmit the data or until the end of the interval indicated by the *l_linger* member, whichever comes first. If SO_LINGER is not specified, and *close*( ) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option shall store a **linger** structure.

SO_OOBINLINE	Reports whether the socket leaves received out-of-band data (data marked urgent) inline. This option shalls tore an **int** value. This is a Boolean option.
SO_SNDBUF	Reports send buffer size information. This option shall store an **int** value.
SO_RCVBUF	Reports receive buffer size information. This option shall store an **int** value.
SO_ERROR	Reports information about error status and clears it. This option shall store an **int** value.
SO_TYPE	Reports the socket type. This option shall store an **int** value. Socket types are described in Section 2.10.6 (on page 509).
SO_DONTROUTE	Reports whether outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option shall store an **int** value. This is a Boolean option.
SO_RCVLOWAT	Reports the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option shall store an **int** value. Note that not all implementations allow this option to be retrieved.
SO_RCVTIMEO	Reports the timeout value for input operations. This option shall store a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data was received. The default for this option is zero, which indicates that a receive operation shall not time out. Note that not all implementations allow this option to be retrieved.
SO_SNDLOWAT	Reports the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option shall store an **int** value. Note that not all implementations allow this option to be retrieved.
SO_SNDTIMEO	Reports the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data were sent. The default for this option is zero, which indicates that a send operation shall not time out. The option shall store a **timeval** structure. Note that not all implementations allow this option to be retrieved.

For Boolean options, a zero value indicates that the option is disabled and a non-zero value indicates that the option is enabled.

Options at other protocol levels vary in format and name.

The socket in use may require the process to have appropriate privileges to use the *getsockopt*( ) function.

RETURN VALUE

Upon successful completion, *getsockopt*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *getsockopt*( ) function shall fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[EINVAL]	The specified option is invalid at the specified socket level.
[ENOPROTOOPT]	The option is not supported by the protocol.
[ENOTSOCK]	The *socket* argument does not refer to a socket.

The *getsockopt*( ) function may fail if:

[EACCES]	The calling process does not have the appropriate privileges.
[EINVAL]	The socket has been shut down.
[ENOBUFS]	Insufficient resources are available in the system to complete the function.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bind( ), *close*( ), *endprotoent*( ), *setsockopt*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**, **<netinet/in.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The **restrict** keyword is added to the *getsockopt*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

getsubopt — parse suboption arguments from a string

SYNOPSIS

XSI

```
#include <stdlib.h>

int getsubopt(char **optionp, char * const *tokens, char **valuep);
```

DESCRIPTION

The *getsubopt*( ) function shall parse suboption arguments in a flag argument. Such options often result from the use of *getopt*( ).

The *getsubopt*( ) argument *optionp* is a pointer to a pointer to the option argument string. The suboption arguments shall be separated by commas and each may consist of either a single token, or a token-value pair separated by an equal sign.

The *keylistp* argument shall be a pointer to a vector of strings. The end of the vector is identified by a null pointer. Each entry in the vector is one of the possible tokens that might be found in **optionp*. Since commas delimit suboption arguments in *optionp*, they should not appear in any of the strings pointed to by *keylistp*. Similarly, because an equal sign separates a token from its value, the application should not include an equal sign in any of the strings pointed to by *keylistp*.

The *valuep* argument is the address of a value string pointer.

If a comma appears in *optionp*, it shall be interpreted as a suboption separator. After commas have been processed, if there are one or more equal signs in a suboption string, the first equal sign in any suboption string shall be interpreted as a separator between a token and a value. Subsequent equal signs in a suboption string shall be interpreted as part of the value.

If the string at **optionp* contains only one suboption argument (equivalently, no commas), *getsubopt*( ) shall update **optionp* to point to the nul character at the end of the string. Otherwise, it shall isolate the suboption argument by replacing the comma separator with a nul character, and shall update **optionp* to point to the start of the next suboption argument. If the suboption argument has an associated value (equivalently, contains an equal sign), *getsubopt*( ) shall update **valuep* to point to the value's first character. Otherwise, it shall set **valuep* to a null pointer. The calling application may use this information to determine whether the presence or absence of a value for the suboption is an error.

Additionally, when *getsubopt*( ) fails to match the suboption argument with a token in the *keylistp* array, the calling application should decide if this is an error, or if the unrecognized option should be processed in another way.

RETURN VALUE

The *getsubopt*( ) function shall return the index of the matched token string, or −1 if no token strings were matched.

ERRORS

No errors are defined.

EXAMPLES

```
#include <stdio.h>
#include <stdlib.h>

int do_all;
const char *type;
int read_size;
int write_size;
int read_only;

enum
{
    RO_OPTION = 0,
    RW_OPTION,
    READ_SIZE_OPTION,
    WRITE_SIZE_OPTION
};

const char *mount_opts[] =
{
    [RO_OPTION] = "ro",
    [RW_OPTION] = "rw",
    [READ_SIZE_OPTION] = "rsize",
    [WRITE_SIZE_OPTION] = "wsize",
    NULL
};

int
main(int argc, char *argv[])
{
    char *subopts, *value;
    int opt;

    while ((opt = getopt(argc, argv, "at:o:")) != -1)
        switch(opt)
            {
            case ’a’:
                do_all = 1;
                break;
            case ’t’:
                type = optarg;
                break;
            case ’o’:
                subopts = optarg;
                while (*subopts != ’\0’)
                    switch(getsubopt(&subopts, mount_opts, &value))
                {
                case RO_OPTION:
                    read_only = 1;
                    break;
                case RW_OPTION:
                    read_only = 0;
                    break;
                case READ_SIZE_OPTION:
```

```
                    if (value == NULL)
                        abort();
                    read_size = atoi(value);
                    break;
                case WRITE_SIZE_OPTION:
                    if (value == NULL)
                        abort();
                    write_size = atoi(value);
                    break;
                default:
                    /* Unknown suboption. */
                    printf("Unknown suboption ‘%s’\n", value);
                    break;
                }
            break;
        default:
            abort();
        }

    /* Do the real work. */

    return 0;
}
```

Parsing Suboptions

The following example uses the *getsubopt*( ) function to parse a value argument in the *optarg* external variable returned by a call to *getopt*( ).

```
#include <stdlib.h>
...
char *tokens[] = {"HOME", "PATH", "LOGNAME", (char *) NULL };
char *value;
int opt, index;

while ((opt = getopt(argc, argv, "e:")) != -1) {
    switch(opt)  {
    case ’e’ :
        while ((index = getsubopt(&optarg, tokens, &value)) != -1) {
            switch(index) {
...
        }
        break;
...
    }
}
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getopt( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

gettimeofday — get the date and time

SYNOPSIS

XSI
```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);

```

DESCRIPTION

The *gettimeofday*( ) function shall obtain the current time, expressed as seconds and microseconds since the Epoch, and store it in the **timeval** structure pointed to by *tp*. The resolution of the system clock is unspecified. |

If *tzp* is not a null pointer, the behavior is unspecified.

RETURN VALUE

The *gettimeofday*( ) function shall return 0 and no value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctime( ), *ftime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to refer to ‘‘seconds since the Epoch’’ rather than ‘‘seconds since 00:00:00 UTC (Coordinated Universal Time), January 1 1970’’ for consistency with other *time* functions.

The **restrict** keyword is added to the *gettimeofday*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

getuid — get a real user ID

SYNOPSIS

```
#include <unistd.h>

uid_t getuid(void);
```

DESCRIPTION

The *getuid*( ) function shall return the real user ID of the calling process.

RETURN VALUE

The *getuid*( ) function shall always be successful and no return value is reserved to indicate the error. |

ERRORS

No errors are defined.

EXAMPLES

Setting the Effective User ID to the Real User ID

The following example sets the effective user ID and the real user ID of the current process to the real user ID of the caller.

```
#include <unistd.h>
#include <sys/types.h>
...
setreuid(getuid(), getuid());
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid( ), *geteuid*( ), *getgid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

getutxent, getutxid, getutxline — get user accounting database entries

SYNOPSIS

XSI

```
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
```

DESCRIPTION

Refer to *endutxent*( ).

NAME

getwc — get a wide character from a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t getwc(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *getwc*( ) function shall be equivalent to *fgetwc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects. |

RETURN VALUE

Refer to *fgetwc*( ).

ERRORS

Refer to *fgetwc*( ).

EXAMPLES

None.

APPLICATION USAGE

Since it may be implemented as a macro, *getwc*( ) may treat incorrectly a *stream* argument with side effects. In particular, *getwc*(**f*++) does not necessarily work as expected. Therefore, use of this function is not recommended; *fgetwc*( ) should be used instead. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fgetwc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

NAME

getwchar — get a wide character from a stdin stream

SYNOPSIS

```
#include <wchar.h>

wint_t getwchar(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *getwchar*( ) function shall be equivalent to *getwc*(*stdin*).

RETURN VALUE

Refer to *fgetwc*( ).

ERRORS

Refer to *fgetwc*( ).

EXAMPLES

None.

APPLICATION USAGE

If the **wint_t** value returned by *getwchar*( ) is stored into a variable of type **wchar_t** and then compared against the **wint_t** macro WEOF, the result may be incorrect. Only the **wint_t** type is guaranteed to be able to represent any wide character and WEOF.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fgetwc( ), *getwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4. Derived from the MSE working draft.

NAME

getwd — get the current working directory pathname (**LEGACY**)

SYNOPSIS

XSI

```
#include <unistd.h>

char *getwd(char *path_name);
```

DESCRIPTION

The *getwd*( ) function shall determine an absolute pathname of the current working directory of the calling process, and copy a string containing that pathname into the array pointed to by the *path_name* argument.

If the length of the pathname of the current working directory is greater than ({PATH_MAX}+1) including the null byte, *getwd*( ) shall fail and return a null pointer.

RETURN VALUE

Upon successful completion, a pointer to the string containing the absolute pathname of the current working directory shall be returned. Otherwise, *getwd*( ) shall return a null pointer and the contents of the array pointed to by *path_name* are undefined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

For applications portability, the *getcwd*( ) function should be used to determine the current working directory instead of *getwd*( ).

RATIONALE

Since the user cannot specify the length of the buffer passed to *getwd*( ), use of this function is discouraged. The length of a pathname described in {PATH_MAX} is file system-dependent and may vary from one mount point to another, or might even be unlimited. It is possible to overflow this buffer in such a way as to cause applications to fail, or possible system security violations.

It is recommended that the *getcwd*( ) function should be used to determine the current working directory.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

getcwd( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

glob, globfree — generate pathnames matching a pattern

SYNOPSIS

```
#include <glob.h>

int glob(const char *restrict pattern, int flags,
    int(*errfunc)(const char *epath, int eerrno),
    glob_t *restrict pglob);
void globfree(glob_t *pglob);
```

DESCRIPTION

The *glob*( ) function is a pathname generator that shall implement the rules defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation, with optional support for rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.3, Patterns Used for Filename Expansion.

The structure type **glob_t** is defined in **<glob.h>** and includes at least the following members:

Member Type	Member Name	Description
size_t	*gl_pathc*	Count of paths matched by *pattern*.
char **	*gl_pathv*	Pointer to a list of matched pathnames.
size_t	*gl_offs*	Slots to reserve at the beginning of *gl_pathv*.

The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob*( ) function shall match all accessible pathnames against this pattern and develop a list of all pathnames that match. In order to have access to a pathname, *glob*( ) requires search permission on every component of a path except the last, and read permission on each directory of any filename component of *pattern* that contains any of the following special characters: `'*'`, `'?'`, and `'['`.

The *glob*( ) function shall store the number of matched pathnames into *pglob->gl_pathc* and a pointer to a list of pointers to path names into *pglob->gl_pathv*. The pathnames shall be in sort order as defined by the current setting of the *LC_COLLATE* category; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, LC_COLLATE. The first pointer after the last pathname shall be a null pointer. If the pattern does not match any pathnames, the returned number of matched paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-defined.

It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob*( ) function shall allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree*( ) function shall free any space associated with *pglob* from a previous call to *glob*( ).

The *flags* argument is used to control the behavior of *glob*( ). The value of *flags* is a bitwise-inclusive OR of zero or more of the following constants, which are defined in **<glob.h>**:

GLOB_APPEND	Append pathnames generated to the ones from a previous call to *glob*( ).
GLOB_DOOFFS	Make use of *pglob->gl_offs*. If this flag is set, *pglob->gl_offs* is used to specify how many null pointers to add to the beginning of *pglob->gl_pathv*. In other words, *pglob->gl_pathv* shall point to *pglob->gl_offs* null pointers, followed by *pglob->gl_pathc* pathname pointers, followed by a null pointer.
GLOB_ERR	Cause *glob*( ) to return when it encounters a directory that it cannot open or read. Ordinarily, *glob*( ) continues to find matches.

GLOB_MARK	Each pathname that is a directory that matches *pattern* shall have a slash appended.
GLOB_NOCHECK	Supports rule 3 in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13.3, Patterns Used for Filename Expansion. If *pattern* does not match any pathname, then *glob*( ) shall return a list consisting of only *pattern*, and the number of matched pathnames is 1.
GLOB_NOESCAPE	Disable backslash escaping.
GLOB_NOSORT	Ordinarily, *glob*( ) sorts the matching pathnames according to the current setting of the *LC_COLLATE* category, see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, LC_COLLATE . When this flag is used, the order of path names returned is unspecified.

The GLOB_APPEND flag can be used to append a new set of pathnames to those found in a previous call to *glob*( ). The following rules apply to applications when two or more calls to *glob*( ) are made with the same value of *pglob* and without intervening calls to *globfree*( ):

1. The first such call shall not set GLOB_APPEND. All subsequent calls shall set it.
2. All the calls shall set GLOB_DOOFFS, or all shall not set it.
3. After the second call, *pglob->gl_pathv* points to a list containing the following:
   a. Zero or more null pointers, as specified by GLOB_DOOFFS and *pglob->gl_offs*.
   b. Pointers to the pathnames that were in the *pglob->gl_pathv* list before the call, in the same order as before.
   c. Pointers to the new pathnames generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* shall be the total number of pathnames from the two calls.
5. The application can change any of the fields after a call to *glob*( ). If it does, the application shall reset them to the original value before a subsequent call, using the same *pglob* value, to *globfree*( ) or *glob*( ) with the GLOB_APPEND flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a null pointer, *glob*( ) calls (**errfunc*( )) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *eerrno* argument is the value of *errno* from the failure, as set by *opendir*( ), *readdir*( ), or *stat*( ). (Other values may be used to report other errors not explicitly documented for those functions.)

If (**errfunc*( )) is called and returns non-zero, or if the GLOB_ERR flag is set in *flags*, *glob*( ) shall stop the scan and return GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is a null pointer or (**errfunc*( )) returns 0, the error shall be ignored.

The *glob*( ) function shall not fail because of large files.

RETURN VALUE

Upon successful completion, *glob*( ) shall return 0. The argument *pglob->gl_pathc* shall return the number of matched pathnames and the argument *pglob->gl_pathv* shall contain a pointer to a null-terminated list of matched and sorted pathnames. However, if *pglob->gl_pathc* is 0, the content of *pglob->gl_pathv* is undefined.

The *globfree*( ) function shall not return a value.

If *glob*( ) terminates due to an error, it shall return one of the non-zero constants defined in **<glob.h>**. The arguments *pglob*->*gl_pathc* and *pglob*->*gl_pathv* are still set as defined above.

ERRORS

The *glob*( ) function shall fail and return the corresponding value if:

GLOB_ABORTED	The scan was stopped because GLOB_ERR was set or (**errfunc*( )) returned non-zero.
GLOB_NOMATCH	The pattern does not match any existing pathname, and GLOB_NOCHECK was not set in flags.
GLOB_NOSPACE	An attempt to allocate memory failed.

EXAMPLES

One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with *execv*( ), *execve*( ), or *execvp*( ). Suppose, for example, that an application wants to do the equivalent of:

```
ls -l *.c
```

but for some reason:

```
system("ls -l *.c")
```

is not acceptable. The application could obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp("ls", &globbuf.gl_pathv[0]);
```

Using the same example:

```
ls -l *.c *.h
```

could be approximately simulated using GLOB_APPEND as follows:

```
globbuf.gl_offs = 2;
glob("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
...
```

APPLICATION USAGE

This function is not provided for the purpose of enabling utilities to perform pathname expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do pathname expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch*( ).

Note that *gl_pathc* and *gl_pathv* have meaning even if *glob*( ) fails. This allows *glob*( ) to report partial results in the event of an error. However, if *gl_pathc* is 0, *gl_pathv* is unspecified even if *glob*( ) did not return an error.

The GLOB_NOCHECK option could be used when an application wants to expand a pathname if wildcards are specified, but wants to treat the pattern as just a string otherwise. The *sh* utility might use this for option-arguments, for example.

The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted together with the previous pathnames. This mirrors the way that the shell handles pathname expansion when multiple expansions are done on a command line.

Applications that need tilde and parameter expansion should use *wordexp*( ).

RATIONALE

It was claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
new = (char **)malloc((n + pglob->gl_pathc + 1)
       * sizeof(char *));
(void) memcpy(new+n, pglob->gl_pathv,
       pglob->gl_pathc * sizeof(char *));
(void) memset(new, 0, n * sizeof(char *));
free(pglob->gl_pathv);
pglob->gl_pathv = new;
```

However, this assumes that the memory pointed to by *gl_pathv* is a block that was separately created using *malloc*( ). This is not necessarily the case. An application should make no assumptions about how the memory referenced by fields in *pglob* was allocated. It might have been obtained from *malloc*( ) in a large chunk and then carved up within *glob*( ), or it might have been created using a different memory allocator. It is not the intent of the standard developers to specify or imply how the memory used by *glob*( ) is managed.

The GLOB_APPEND flag would be used when an application wants to expand several different patterns into a single list.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fnmatch*( ), *opendir*( ), *readdir*( ), *stat*( ), *wordexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<glob.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *glob*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

gmtime, gmtime_r — convert a time value to a broken-down UTC time

SYNOPSIS

```
#include <time.h>

struct tm *gmtime(const time_t *timer);
struct tm *gmtime_r(const time_t *restrict timer,
    struct tm *restrict result);
```

TSF

DESCRIPTION

CX For *gmtime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *gmtime*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (UTC).

CX The relationship between a time in seconds since the Epoch used as an argument to *gmtime*( ) and the **tm** structure (defined in the **<time.h>** header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch), where the names in the structure and in the expression correspond.

TSF The same relationship shall apply for *gmtime_r*( ).

CX The *gmtime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

TSF The *gmtime_r*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure referred to by *result*. The *gmtime_r*( ) function shall also return the address of the same structure.

RETURN VALUE

The *gmtime*( ) function shall return a pointer to a **struct tm**.

TSF Upon successful completion, *gmtime_r*( ) shall return the address of the structure pointed to by the argument *result*. If an error is detected, or UTC is not available, *gmtime_r*( ) shall return a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *gmtime_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *gmtime*( ) function need not be reentrant is added to the DESCRIPTION.

The *gmtime_r*( ) function is included for alignment with the POSIX Threads Extension.

Issue 6

The *gmtime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *gmtime_r*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

grantpt — grant access to the slave pseudo-terminal device

SYNOPSIS

XSI

```
#include <stdlib.h>

int grantpt(int fildes);
```

DESCRIPTION

The *grantpt*( ) function shall change the mode and ownership of the slave pseudo-terminal device associated with its master pseudo-terminal counterpart. The *fildes* argument is a file descriptor that refers to a master pseudo-terminal device. The user ID of the slave shall be set to the real UID of the calling process and the group ID shall be set to an unspecified group ID. The permission mode of the slave pseudo-terminal shall be set to readable and writable by the owner, and writable by the group.

The behavior of the *grantpt*( ) function is unspecified if the application has installed a signal handler to catch SIGCHLD signals.

RETURN VALUE

Upon successful completion, *grantpt*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *grantpt*( ) function may fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EINVAL]	The *fildes* argument is not associated with a master pseudo-terminal device.
[EACCES]	The corresponding slave pseudo-terminal device could not be accessed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section in previous issues.

NAME

h_errno — error return value for network database operations

SYNOPSIS

OB

```
#include <netdb.h>
```

DESCRIPTION

Note that this method of returning errors is used only in connection with obsolescent functions.

The **<netdb.h>** header provides a declaration of *h_errno* as a modifiable *l*-value of type **int**.

It is unspecified whether *h_errno* is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name *h_errno*, the behavior is undefined.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications should obtain the definition of *h_errno* by the inclusion of the **<netdb.h>** header.

RATIONALE

None.

FUTURE DIRECTIONS

h_errno may be withdrawn in a future version.

SEE ALSO

endhostent( ), *errno*, the Base Definitions volume of IEEE Std 1003.1-200x, **<netdb.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

hcreate, hdestroy, hsearch — manage hash search table

SYNOPSIS

XSI

```
#include <search.h>

int hcreate(size_t nel);
void hdestroy(void);
ENTRY *hsearch(ENTRY item, ACTION action);
```

DESCRIPTION

The *hcreate*( ), *hdestroy*( ), and *hsearch*( ) functions shall manage hash search tables. |

The *hcreate*( ) function shall allocate sufficient space for the table, and the application shall | ensure it is called before *hsearch*( ) is used. The *nel* argument is an estimate of the maximum | number of entries that the table shall contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The *hdestroy*( ) function shall dispose of the search table, and may be followed by another call to | *hcreate*( ). After the call to *hdestroy*( ), the data can no longer be considered accessible.

The *hsearch*( ) function is a hash-table search routine. It shall return a pointer into a hash table indicating the location at which an entry can be found. The *item* argument is a structure of type **ENTRY** (defined in the **<search.h>** header) containing two pointers: *item.key* points to the comparison key (a **char ***), and *item.data* (a **void ***) points to any other data to be associated with that key. The comparison function used by *hsearch*( ) is *strcmp*( ). The *action* argument is a member of an enumeration type **ACTION** indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *hcreate*( ) function shall return 0 if it cannot allocate sufficient space for the table; otherwise, it shall return non-zero.

The *hdestroy*( ) function shall not return a value.

The *hsearch*( ) function shall return a null pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full.

ERRORS

The *hcreate*( ) and *hsearch*( ) functions may fail if:

[ENOMEM]	Insufficient storage space is available.

EXAMPLES

The following example reads in strings followed by two numbers and stores them in a hash table, discarding duplicates. It then reads in strings and finds the matching entry in the hash table and prints it out.

```
#include <stdio.h>
#include <search.h>
#include <string.h>

struct info {        /* This is the info stored in the table */
    int age, room;   /* other than the key. */
```

```
};
#define NUM_EMPL    5000    /* # of elements in search table. */
int main(void)
{
    char string_space[NUM_EMPL*20];   /* Space to store strings. */
    struct info info_space[NUM_EMPL]; /* Space to store employee info. */
    char *str_ptr = string_space;     /* Next space in string_space. */
    struct info *info_ptr = info_space;
                                      /* Next space in info_space. */
    ENTRY item;
    ENTRY *found_item; /* Name to look for in table. */
    char name_to_find[30];
    int i = 0;
    /* Create table; no error checking is performed. */
    (void) hcreate(NUM_EMPL);
    while (scanf("%s%d%d", str_ptr, &info_ptr->age,
           &info_ptr->room) != EOF && i++ < NUM_EMPL) {
        /* Put information in structure, and structure in item. */
        item.key = str_ptr;
        item.data = info_ptr;
        str_ptr += strlen(str_ptr) + 1;
        info_ptr++;
        /* Put item into table. */
        (void) hsearch(item, ENTER);
    }
    /* Access table. */
    item.key = name_to_find;
    while (scanf("%s", item.key) != EOF) {
        if ((found_item = hsearch(item, FIND)) != NULL) {
            /* If item is in the table. */
            (void)printf("found %s, age = %d, room = %d\n",
                found_item->key,
                ((struct info *)found_item->data)->age,
                ((struct info *)found_item->data)->room);
        } else
            (void)printf("no such employee %s\n", name_to_find);
    }
    return 0;
}
```

APPLICATION USAGE

The *hcreate*( ) and *hsearch*( ) functions may use *malloc*( ) to allocate space.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bsearch( ), *lsearch*( ), *malloc*( ), *strcmp*( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

hdestroy — manage hash search table

SYNOPSIS

XSI

```
#include <search.h>

void hdestroy(void);

```

DESCRIPTION

Refer to *hcreate*( ).

NAME

hsearch — manage hash search table

SYNOPSIS

XSI

```
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);

```

DESCRIPTION

Refer to *hcreate*( ).

NAME

htonl, htons, ntohl, ntohs — convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

DESCRIPTION

These functions shall convert 16-bit and 32-bit quantities between network byte order and host byte order.

On some implementations, these functions are defined as macros.

The **uint32_t** and **uint16_t** types are defined in **<inttypes.h>**.

RETURN VALUE

The *htonl*( ) and *htons*( ) functions shall return the argument value converted from host to network byte order.

The *ntohl*( ) and *ntohs*( ) functions shall return the argument value converted from network to host byte order.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

These functions are most often used in conjunction with IPv4 addresses and ports as returned by *gethostent*( ) and *getservent*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endhostent( ), *endservent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**, **<arpa/inet.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

htons — convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

uint16_t htons(uint16_t hostshort);
```

DESCRIPTION

Refer to *htonl*( ).

NAME

hypot, hypotf, hypotl — Euclidean distance function

SYNOPSIS

```
#include <math.h>

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the value of the square root of x^2+y^2 without undue overflow or underflow.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the length of the hypotenuse of a right-angled triangle with sides of length *x* and *y*.

If the correct value would cause overflow, a range error shall occur and *hypot*( ), *hypotf*( ), and *hypotl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

MX If *x* or *y* is ±Inf, +Inf shall be returned (even if one of *x* or *y* is NaN).

If *x* or *y* is NaN, and the other is not ±Inf, a NaN shall be returned.

If both arguments are subnormal and the correct result is subnormal, a range error may occur and the correct result is returned.

ERRORS

These functions shall fail if:

Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

MX Range Error The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

hypot(*x*,*y*), *hypot*(*y*,*x*), and *hypot*(*x*, −*y*) are equivalent.

hypot(*x*, ±0) is equivalent to *fabs*(*x*).

Underflow only happens when both *x* and *y* are subnormal and the (inexact) result is also subnormal.

These functions take precautions against overflow during intermediate steps of the computation.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *sqrt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *hypot*( ) function is no longer marked as an extension.

The *hypotf*( ) and *hypotl*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

NAME

iconv — codeset conversion function

SYNOPSIS

XSI

```
#include <iconv.h>

size_t iconv(iconv_t cd, char **restrict inbuf,
    size_t *restrict inbytesleft, char **restrict outbuf,
    size_t *restrict outbytesleft);
```

DESCRIPTION

The *iconv*( ) function shall convert the sequence of characters from one codeset, in the array specified by *inbuf*, into a sequence of corresponding characters in another codeset, in the array specified by *outbuf*. The codesets are those specified in the *iconv_open*( ) call that returned the conversion descriptor, *cd*. The *inbuf* argument points to a variable that points to the first character in the input buffer and *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first available byte in the output buffer and *outbytesleft* indicates the number of the available bytes to the end of the buffer.

For state-dependent encodings, the conversion descriptor *cd* is placed into its initial shift state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When *iconv*( ) is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, *iconv*( ) shall place, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, *iconv*( ) shall fail and set *errno* to [E2BIG]. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified codeset, conversion shall stop after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion shall stop after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion shall stop just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by *inbuf* shall be updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by *inbytesleft* shall be decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* shall be updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* shall be decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor shall be updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If *iconv*( ) encounters a character in the input buffer that is valid, but for which an identical character does not exist in the target codeset, *iconv*( ) shall perform an implementation-defined conversion on this character.

RETURN VALUE

The *iconv*( ) function shall update the variables pointed to by the arguments to reflect the extent of the conversion and return the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* shall be 0. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by *inbytesleft* shall be non-zero and *errno* shall be set to indicate the condition. If an error occurs *iconv*( ) shall return (**size_t**)–1 and set *errno* to indicate the error.

ERRORS

The *iconv*( ) function shall fail if:

[EILSEQ] Input conversion stopped due to an input byte that does not belong to the input codeset.

[E2BIG] Input conversion stopped due to lack of space in the output buffer.

[EINVAL] Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The *iconv*( ) function may fail if:

[EBADF] The *cd* argument is not a valid open conversion descriptor.

EXAMPLES

None.

APPLICATION USAGE

The *inbuf* argument indirectly points to the memory area which contains the conversion input data. The *outbuf* argument indirectly points to the memory area which is to contain the result of the conversion. The objects indirectly pointed to by *inbuf* and *outbuf* are not restricted to containing data that is directly representable in the ISO C standard language **char** data type. The type of *inbuf* and *outbuf*, **char ****, does not imply that the objects pointed to are interpreted as null-terminated C strings or arrays of characters. Any interpretation of a byte sequence that represents a character in a given character set encoding scheme is done internally within the codeset converters. For example, the area pointed to indirectly by *inbuf* and/or *outbuf* can contain all zero octets that are not interpreted as string terminators but as coded character data according to the respective codeset encoding scheme. The type of the data (**char**, **short**, **long**, and so on) read or stored in the objects is not specified, but may be inferred for both the input and output data by the converters determined by the *fromcode* and *tocode* arguments of *iconv_open*( ).

Regardless of the data type inferred by the converter, the size of the remaining space in both input and output objects (the *intbytesleft* and *outbytesleft* arguments) is always measured in bytes.

For implementations that support the conversion of state-dependent encodings, the conversion descriptor must be able to accurately reflect the shift-state in effect at the end of the last successful conversion. It is not required that the conversion descriptor itself be updated, which would require it to be a pointer type. Thus, implementations are free to implement the descriptor as a handle (other than a pointer type) by which the conversion information can be accessed and updated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iconv_open( ), *iconv_close*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<iconv.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the HP-UX Manual.

Issue 6

The SYNOPSIS has been corrected to align with the **<iconv.h>** reference page.

The **restrict** keyword is added to the *iconv*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

iconv_close — codeset conversion deallocation function

SYNOPSIS

XSI
```
#include <iconv.h>

int iconv_close(iconv_t cd);
```

DESCRIPTION

The *iconv_close*( ) function shall deallocate the conversion descriptor *cd* and all other associated resources allocated by *iconv_open*( ). |

If a file descriptor is used to implement the type **iconv_t**, that file descriptor shall be closed. |

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *iconv_close*( ) function may fail if:

[EBADF]	The conversion descriptor is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iconv( ), *iconv_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<iconv.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the HP-UX Manual.

NAME

iconv_open — codeset conversion allocation function

SYNOPSIS

XSI

```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);
```

DESCRIPTION

The *iconv_open*( ) function shall return a conversion descriptor that describes a conversion from the codeset specified by the string pointed to by the *fromcode* argument to the codeset specified by the string pointed to by the *tocode* argument. For state-dependent encodings, the conversion descriptor shall be in a codeset-dependent initial shift state, ready for immediate use with *iconv*( ). |

Settings of *fromcode* and *tocode* and their permitted combinations are implementation-defined.

A conversion descriptor shall remain valid until it is closed by *iconv_close*( ) or an implicit close. |

If a file descriptor is used to implement conversion descriptors, the FD_CLOEXEC flag shall be set; see **<fcntl.h>**.

RETURN VALUE

Upon successful completion, *iconv_open*( ) shall return a conversion descriptor for use on subsequent calls to *iconv*( ). Otherwise, *iconv_open*( ) shall return (**iconv_t**)–1 and set *errno* to indicate the error.

ERRORS

The *iconv_open*( ) function may fail if:

[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	Too many files are currently open in the system.
[ENOMEM]	Insufficient storage space is available.
[EINVAL]	The conversion specified by *fromcode* and *tocode* is not supported by the implementation.

EXAMPLES

None.

APPLICATION USAGE

Some implementations of *iconv_open*( ) use *malloc*( ) to allocate space for internal buffer areas. The *iconv_open*( ) function may fail if there is insufficient storage space to accommodate these buffers.

Conforming applications must assume that conversion descriptors are not valid after a call to one of the *exec* functions. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iconv( ), *iconv_close*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<iconv.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the HP-UX Manual.

NAME

if_freenameindex — free memory allocated by *if_nameindex*( )

SYNOPSIS

```
#include <net/if.h>

void if_freenameindex(struct if_nameindex *ptr);
```

DESCRIPTION

The *if_freenameindex*( ) function shall free the memory allocated by *if_nameindex*( ). The *ptr* argument shall be a pointer that was returned by *if_nameindex*( ). After *if_freenameindex*( ) has been called, the application shall not use the array of which *ptr* is the address.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *if_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

if_indextoname — map a network interface index to its corresponding name

SYNOPSIS

```
#include <net/if.h>

char *if_indextoname(unsigned ifindex, char *ifname);
```

DESCRIPTION

The *if_indextoname*( ) function shall map an interface index to its corresponding name.

When this function is called, *ifname* shall point to a buffer of at least {IFNAMSIZ} bytes. The function shall place in this buffer the name of the interface with index *ifindex*.

RETURN VALUE

If *ifindex* is an interface index, then the function shall return the value supplied in *ifname*, which points to a buffer now containing the interface name. Otherwise, the function shall return a NULL pointer and set *errno* to indicate the error.

ERRORS

The *if_indextoname*( ) function shall fail if:

[ENXIO] The interface does not exist.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *if_freenameindex*( ), *if_nameindex*( ), *if_nametoindex*( ), *setsockopt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

if_nameindex — return all network interface names and indexes

SYNOPSIS

```
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
```

DESCRIPTION

The *if_nameindex*( ) function shall return an array of *if_nameindex* structures, one structure per interface. The end of the array is indicated by a structure with an *if_index* field of zero and an *if_name* field of NULL.

Applications should call *if_freenameindex*( ) to release the memory that may be dynamically allocated by this function, after they have finished using it.

RETURN VALUE

Array of structures identifying local interfaces. A NULL pointer is returned upon an error, with *errno* set to indicate the error.

ERRORS

The *if_nameindex*( ) function may fail if:

[ENOBUFS]	Insufficient resources are available to complete the function.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nametoindex*( ), *setsockopt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

if_nametoindex — map a network interface name to its corresponding index

SYNOPSIS

```
#include <net/if.h>

unsigned if_nametoindex(const char *ifname);
```

DESCRIPTION

The *if_nametoindex*( ) function shall return the interface index corresponding to name *ifname*.

RETURN VALUE

The corresponding index if *ifname* is the name of an interface; otherwise, zero.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *setsockopt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<net/if.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

ilogb, ilogbf, ilogbl — return an unbiased exponent

SYNOPSIS

```
#include <math.h>

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall return the exponent part of their argument *x*. Formally, the return value is the integral part of $\log_r | x |$ as a signed integral value, for non-zero *x*, where *r* is the radix of the machine's floating-point arithmetic, which is the value of FLT_RADIX defined in **<float.h>**.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the exponent part of *x* as a signed integer value. They are equivalent to calling the corresponding *logb*( ) function and casting the returned value to type **int**.

XSI If *x* is 0, a domain error shall occur, and the value FP_ILOGB0 shall be returned.

XSI If *x* is ±Inf, a domain error shall occur, and the value {INT_MAX} shall be returned.

XSI If *x* is a NaN, a domain error shall occur, and the value FP_ILOGBNAN shall be returned.

XSI If the correct value is greater than {INT_MAX}, {INT_MAX} shall be returned and a domain error shall occur.

If the correct value is less than {INT_MIN}, {INT_MIN} shall be returned and a domain error shall occur.

ERRORS

These functions shall fail if:

XSI Domain Error The *x* argument is zero, NaN, or ±Inf, or the correct value is not representable as an integer.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

The errors come from taking the expected floating-point value and converting it to **int**, which is invalid operation in IEEE Std 754-1985 (since overflow, infinity, and NaN are not representable in a type **int**), so should be a domain error.

There are no known implementations that overflow. For overflow to happen, {INT_MAX} must be less than LDBL_MAX_EXP**log2*(FLT_RADIX) or {INT_MIN} must be greater than LDBL_MIN_EXP**log2*(FLT_RADIX) if subnormals are not supported, or {INT_MIN} must be greater than (LDBL_MIN_EXP-LDBL_MANT_DIG)**log2*(FLT_RADIX) if subnormals are supported.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *logb*( ), *scalb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *ilogb*( ) function is no longer marked as an extension.

The *ilogbf*( ) and *ilogbl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The RETURN VALUE section is revised for alignment with the ISO/IEC 9899:1999 standard.

XSI extensions are marked.

NAME

imaxabs — return absolute value

SYNOPSIS

```
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *imaxabs*( ) function shall compute the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.

RETURN VALUE

The *imaxabs*( ) function shall return the absolute value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The absolute value of the most negative number cannot be represented in two's complement.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

imaxdiv( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

imaxdiv — return quotient and remainder

SYNOPSIS

```
#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *imaxdiv*( ) function shall compute *numer* / *denom* and *numer* % *denom* in a single operation.

RETURN VALUE

The *imaxdiv*( ) function shall return a structure of type **imaxdiv_t**, comprising both the quotient and the remainder. The structure shall contain (in either order) the members *quot* (the quotient) and *rem* (the remainder), each of which has type **intmax_t**.

If either part of the result cannot be represented, the behavior is undefined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

imaxabs( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

index — character string operations (**LEGACY**)

SYNOPSIS

XSI

```
#include <strings.h>

char *index(const char *s, int c);
```

DESCRIPTION

The *index*( ) function shall be equivalent to *strchr*( ).

RETURN VALUE

See *strchr*( ).

ERRORS

See *strchr*( ).

EXAMPLES

None.

APPLICATION USAGE

strchr( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *index*( ) as follows:

```
#define index(a,b) strchr((a),(b))
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

strchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X / OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

inet_addr, inet_ntoa — IPv4 address manipulation

SYNOPSIS

```
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);
```

DESCRIPTION

The *inet_addr*( ) function shall convert the string pointed to by *cp*, in the standard IPv4 dotted decimal notation, to an integer value suitable for use as an Internet address.

The *inet_ntoa*( ) function shall convert the Internet host address specified by *in* to a string in the Internet standard dot notation.

The *inet_ntoa*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

All Internet addresses shall be returned in network order (bytes ordered from left to right).

Values specified using IPv4 dotted decimal notation take one of the following forms:

`a.b.c.d`	When four parts are specified, each shall be interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.
`a.b.c`	When a three-part address is specified, the last part shall be interpreted as a 16-bit quantity and placed in the rightmost two bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as **128.net.host**.
`a.b`	When a two-part address is supplied, the last part shall be interpreted as a 24-bit quantity and placed in the rightmost three bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as **net.host**.
`a`	When only one part is given, the value shall be stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in IPv4 dotted decimal notation may be decimal, octal, or hexadecimal, as specified in the ISO C standard (that is, a leading 0x or 0X implies hexadecimal; otherwise, a leading `'0'` implies octal; otherwise, the number is interpreted as decimal).

RETURN VALUE

Upon successful completion, *inet_addr*( ) shall return the Internet address. Otherwise, it shall return (**in_addr_t**)(−1).

The *inet_ntoa*( ) function shall return a pointer to the network address in Internet standard dot notation.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The return value of *inet_ntoa*( ) may point to static data that may be overwritten by subsequent calls to *inet_ntoa*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

endhostent( ), *endnetent*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

inet_ntoa — IPv4 address manipulation

SYNOPSIS

```
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

DESCRIPTION

Refer to *inet_addr*( ).

NAME

inet_ntop, inet_pton — convert IPv4 and IPv6 addresses between binary and text form

SYNOPSIS

```
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *restrict src,
    char *restrict dst, socklen_t size);
int inet_pton(int af, const char *restrict src, void *restrict dst);
```

DESCRIPTION

The *inet_ntop*( ) function shall convert a numeric address into a text string suitable for presentation. The *af* argument shall specify the family of the address. This can be AF_INET or AF_INET6. The *src* argument points to a buffer holding an IPv4 address if the *af* argument is AF_INET, or an IPv6 address if the *af* argument is AF_INET6. The *dst* argument points to a buffer where the function stores the resulting text string; it shall not be NULL. The *size* argument specifies the size of this buffer, which shall be large enough to hold the text string (INET_ADDRSTRLEN characters for IPv4, INET6_ADDRSTRLEN characters for IPv6).

The *inet_pton*( ) function shall convert an address in its standard text presentation form into its numeric binary form. The *af* argument shall specify the family of the address. The AF_INET and AF_INET6 address families shall be supported. The *src* argument points to the string being passed in. The *dst* argument points to a buffer into which the function stores the numeric address; this shall be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6).

If the *af* argument of *inet_pton*( ) is AF_INET, the *src* string shall be in the standard IPv4 dotted-decimal form:

```
ddd.ddd.ddd.ddd
```

where `"ddd"` is a one to three digit decimal number between 0 and 255 (see *inet_addr*( )). The *inet_pton*( ) function does not accept other formats (such as the octal numbers, hexadecimal numbers, and fewer than four numbers that *inet_addr*( ) accepts).

If the *af* argument of *inet_pton*( ) is AF_INET6, the *src* string shall be in one of the following standard IPv6 text forms:

1. The preferred form is `"x:x:x:x:x:x:x:x"`, where the `'x'`s are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there shall be at least one numeral in every field.
2. A string of contiguous zero fields in the preferred form can be shown as `"::"`. The `"::"` can only appear once in an address. Unspecified addresses (`"0:0:0:0:0:0:0:0"`) may be represented simply as `"::"`.
3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 nodes is `"x:x:x:x:x:x:d.d.d.d"`, where the `'x'`s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the `'d'`s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

Note: A more extensive description of the standard representations of IPv6 addresses can be found in RFC 2373.

RETURN VALUE

The *inet_ntop*( ) function shall return a pointer to the buffer containing the text string if the conversion succeeds, and NULL otherwise, and set *errno* to indicate the error.

The *inet_pton*( ) function shall return 1 if the conversion succeeds, with the address pointed to by *dst* in network byte order. It shall return 0 if the input is not a valid IPv4 dotted-decimal string IP6 or a valid IPv6 address string, or −1 with *errno* set to [EAFNOSUPPORT] if the *af* argument is unknown.

ERRORS

The *inet_ntop*( ) and *inet_pton*( ) functions shall fail if:

[EAFNOSUPPORT]	The *af* argument is invalid.
[ENOSPC]	The size of the *inet_ntop*( ) result buffer is inadequate.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<arpa/inet.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

IPv6 extensions are marked.

The **restrict** keyword is added to the *inet_ntop*( ) and *inet_pton*( ) prototypes for alignment with the ISO / IEC 9899: 1999 standard.

NAME

initstate, random, setstate, srandom — pseudo-random number functions

SYNOPSIS

XSI

```
#include <stdlib.h>

char *initstate(unsigned seed, char *state, size_t size);
long random(void);
char *setstate(const char *state);
void srandom(unsigned seed);

```

DESCRIPTION

The *random*( ) function shall use a non-linear additive feedback random-number generator employing a default state array size of 31 **long** integers to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$. The period of this random-number generator is approximately 16 x ($2^{31}-1$). The size of the state array determines the period of the random-number generator. Increasing the state array size shall increase the period.

With 256 bytes of state information, the period of the random-number generator shall be greater than 2^{69}.

Like *rand*( ), *random*( ) shall produce by default a sequence of numbers that can be duplicated by calling *srandom*( ) with 1 as the seed.

The *srandom*( ) function shall initialize the current state array using the value of *seed*.

The *initstate*( ) and *setstate*( ) functions handle restarting and changing random-number generators. The *initstate*( ) function allows a state array, pointed to by the *state* argument, to be initialized for future use. The *size* argument, which specifies the size in bytes of the state array, shall be used by *initstate*( ) to decide what type of random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Other values greater than 8 bytes are rounded down to the nearest one of these values. If *initstate*( ) is called with $8 \leq size < 32$, then *random*( ) shall use a simple linear congruential random number generator. The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The *initstate*( ) function shall return a pointer to the previous state information array.

If *initstate*( ) has not been called, then *random*( ) shall behave as though *initstate*( ) had been called with *seed*=1 and *size*=128.

Once a state has been initialized, *setstate*( ) allows switching between state arrays. The array defined by the *state* argument shall be used for further random-number generation until *initstate*( ) is called or *setstate*( ) is called again. The *setstate*( ) function shall return a pointer to the previous state array.

RETURN VALUE

If *initstate*( ) is called with *size* less than 8, it shall return NULL.

The *random*( ) function shall return the generated pseudo-random number.

The *srandom*( ) function shall not return a value.

Upon successful completion, *initstate*( ) and *setstate*( ) shall return a pointer to the previous state array; otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

After initialization, a state array can be restarted at a different point in one of two ways:

1. The *initstate*( ) function can be used, with the desired seed, state array, and size of the array.
2. The *setstate*( ) function, with the desired state, can be used, followed by *srandom*( ) with the desired seed. The advantage of using both of these functions is that the size of the state array does not have to be saved once it is initialized.

Although some implementations of *random*( ) have written messages to standard error, such implementations do not conform to this volume of IEEE Std 1003.1-200x.

Issue 5 restores the historical behavior of this function.

Threaded applications should use *rand_r*( ), *erand48*( ), *nrand48*( ), or *jrand48*( ) instead of *random*( ) when an independent random number sequence in multiple threads is required.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

drand48( ), *rand*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the phrase ‘‘values smaller than 8’’ is replaced with ‘‘values greater than or equal to 8, or less than 32’’, ‘‘*size*<8’’ is replaced with ‘‘8≤*size* <32’’, and a new first paragraph is added to the RETURN VALUE section. A note is added to the APPLICATION USAGE indicating that these changes restore the historical behavior of the function.

Issue 6

In the DESCRIPTION, duplicate text ‘‘For values greater than or equal to 8 . . .’’ is removed.

NAME

insque, remque — insert or remove an element in a queue

SYNOPSIS

XSI

```
#include <search.h>

void insque(void *element, void *pred);
void remque(void *element);
```

DESCRIPTION

The *insque*( ) and *remque*( ) functions shall manipulate queues built from doubly-linked lists. The queue can be either circular or linear. An application using *insque*( ) or *remque*( ) shall ensure it defines a structure in which the first two members of the structure are pointers to the same type of structure, and any further members are application-specific. The first member of the structure is a forward pointer to the next entry in the queue. The second member is a backward pointer to the previous entry in the queue. If the queue is linear, the queue is terminated with null pointers. The names of the structure and of the pointer members are not subject to any special restriction. |

The *insque*( ) function shall insert the element pointed to by *element* into a queue immediately after the element pointed to by *pred*. |

The *remque*( ) function shall remove the element pointed to by *element* from a queue. |

If the queue is to be used as a linear list, invoking *insque*(&*element*, NULL), where *element* is the initial element of the queue, shall initialize the forward and backward pointers of *element* to null pointers.

If the queue is to be used as a circular list, the application shall ensure it initializes the forward pointer and the backward pointer of the initial element of the queue to the element's own address.

RETURN VALUE

The *insque*( ) and *remque*( ) functions do not return a value.

ERRORS

No errors are defined.

EXAMPLES

Creating a Linear Linked List

The following example creates a linear linked list.

```
#include <search.h>
...
struct myque element1;
struct myque element2;

char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;

insque (&element1, NULL);
insque (&element2, &element1);
```

Creating a Circular Linked List

The following example creates a circular linked list.

```
#include <search.h>
...
struct myque element1;
struct myque element2;

char *data1 = "DATA1";
char *data2 = "DATA2";
...
element1.data = data1;
element2.data = data2;

element1.fwd = &element1;
element1.bck = &element1;

insque (&element2, &element1);
```

Removing an Element

The following example removes the element pointed to by *element1*.

```
#include <search.h>
...
struct myque element1;
...
remque (&element1);
```

APPLICATION USAGE

The historical implementations of these functions described the arguments as being of type **struct qelem** * rather than as being of type **void** * as defined here. In those implementations, **struct qelem** was commonly defined in **<search.h>** as:

```
struct qelem {
    struct qelem  *q_forw;
    struct qelem  *q_back;
};
```

Applications using these functions, however, were never able to use this structure directly since it provided no room for the actual data contained in the elements. Most applications defined structures that contained the two pointers as the initial elements and also provided space for, or pointers to, the object's data. Applications that used these functions to update more than one type of table also had the problem of specifying two or more different structures with the same name, if they literally used **struct qelem** as specified.

As described here, the implementations were actually expecting a structure type where the first two members were forward and backward pointers to structures. With C compilers that didn't provide function prototypes, applications used structures as specified in the DESCRIPTION above and the compiler did what the application expected.

If this method had been carried forward with an ISO C standard compiler and the historical function prototype, most applications would have to be modified to cast pointers to the structures actually used to be pointers to **struct qelem** to avoid compilation warnings. By specifying **void** * as the argument type, applications do not need to change (unless they specifically referenced **struct qelem** and depended on it being defined in **<search.h>**).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

ioctl — control a STREAMS device (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>

int ioctl(int fildes, int request, ... /* arg */);
```

DESCRIPTION

The *ioctl*( ) function shall perform a variety of control functions on STREAMS devices. For non-STREAMS devices, the functions performed by this call are unspecified. The *request* argument and an optional third argument (with varying type) shall be passed to and interpreted by the appropriate part of the STREAM associated with *fildes*.

The *fildes* argument is an open file descriptor that refers to a device.

The *request* argument selects the control function to be performed and shall depend on the STREAMS device being addressed.

The *arg* argument represents additional information that is needed by this specific STREAMS device to perform the requested function. The type of *arg* depends upon the particular control request, but it shall be either an integer or a pointer to a device-specific data structure.

The *ioctl*( ) commands applicable to STREAMS, their arguments, and error conditions that apply to each individual command are described below.

The following *ioctl*( ) commands, with error values indicated, are applicable to all STREAMS files:

I_PUSH Pushes the module whose name is pointed to by *arg* onto the top of the current STREAM, just below the STREAM head. It then calls the *open*( ) function of the newly-pushed module.

The *ioctl*( ) function with the I_PUSH command shall fail if:

[EINVAL]	Invalid module name.
[ENXIO]	Open function of new module failed.
[ENXIO]	Hangup received on *fildes*.

I_POP Removes the module just below the STREAM head of the STREAM pointed to by *fildes*. The *arg* argument should be 0 in an I_POP request.

The *ioctl*( ) function with the I_POP command shall fail if:

[EINVAL]	No module present in the STREAM.
[ENXIO]	Hangup received on *fildes*.

I_LOOK Retrieves the name of the module just below the STREAM head of the STREAM pointed to by *fildes*, and places it in a character string pointed to by *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long, where FMNAMESZ is defined in **<stropts.h>**.

The *ioctl*( ) function with the I_LOOK command shall fail if:

[EINVAL]	No module present in the STREAM.

I_FLUSH Flushes read and/or write queues, depending on the value of *arg*. Valid *arg* values are:

FLUSHR	Flush all read queues.
FLUSHW	Flush all write queues.
FLUSHRW	Flush all read and all write queues.

The *ioctl*( ) function with the I_FLUSH command shall fail if:

[EINVAL]	Invalid *arg* value.
[EAGAIN] or [ENOSR]	Unable to allocate buffers for flush message.
[ENXIO]	Hangup received on *fildes*.

I_FLUSHBAND	Flushes a particular band of messages. The *arg* argument points to a **bandinfo** structure. The *bi_flag* member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The *bi_pri* member determines the priority band to be flushed.
I_SETSIG	Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with *fildes*. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-inclusive OR of any combination of the following constants:

S_RDNORM	A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
S_RDBAND	A message with a non-zero priority band has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
S_INPUT	A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal shall be generated even if the message is of zero length.
S_HIPRI	A high-priority message is present on a STREAM head read queue. A signal shall be generated even if the message is of zero length.
S_OUTPUT	The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
S_WRNORM	Equivalent to S_OUTPUT.
S_WRBAND	The write queue for a non-zero priority band just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) priority data downstream.
S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
S_ERROR	Notification of an error condition has reached the STREAM head.

S_HANGUP	Notification of a hangup has reached the STREAM head.
S_BANDURG	When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.

If *arg* is 0, the calling process shall be unregistered and shall not receive further SIGPOLL signals for the stream associated with *fildes*.

Processes that wish to receive SIGPOLL signals shall ensure that they explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same STREAM, each process shall be signaled when the event occurs.

The *ioctl*( ) function with the I_SETSIG command shall fail if:

[EINVAL]	The value of *arg* is invalid.
[EINVAL]	The value of *arg* is 0 and the calling process is not registered to receive the SIGPOLL signal.
[EAGAIN]	There were insufficient resources to store the signal request.

I_GETSIG

Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an **int** pointed to by *arg*, where the events are those specified in the description of I_SETSIG above.

The *ioctl*( ) function with the I_GETSIG command shall fail if:

[EINVAL]	Process is not registered to receive the SIGPOLL signal.

I_FIND

Compares the names of all modules currently present in the STREAM to the name pointed to by *arg*, and returns 1 if the named module is present in the STREAM, or returns 0 if the named module is not present.

The *ioctl*( ) function with the I_FIND command shall fail if:

[EINVAL]	*arg* does not contain a valid module name.

I_PEEK

Retrieves the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to *getmsg*( ) except that this command does not remove the message from the queue. The *arg* argument points to a **strpeek** structure.

The application shall ensure that the *maxlen* member in the **ctlbuf** and **databuf strbuf** structures is set to the number of bytes of control information and/or data information, respectively, to retrieve. The *flags* member may be marked RS_HIPRI or 0, as described by *getmsg*( ). If the process sets *flags* to RS_HIPRI, for example, I_PEEK shall only look for a high-priority message on the STREAM head read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPRI flag was set in *flags* and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, **ctlbuf** specifies information in the control buffer, **databuf** specifies information in the data buffer, and *flags* contains the value RS_HIPRI or 0.

I_SRDOPT

Sets the read mode using the value of the argument *arg*. Read modes are described in *read*( ). Valid *arg* flags are:

RNORM	Byte-stream mode, the default.
RMSGD	Message-discard mode.
RMSGN	Message-nondiscard mode.

The bitwise-inclusive OR of RMSGD and RMSGN shall return [EINVAL]. The bitwise-inclusive OR of RNORM and either RMSGD or RMSGN shall result in the other flag overriding RNORM which is the default.

In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flags in *arg*:

RPROTNORM	Fail *read*( ) with [EBADMSG] if a message containing a control part is at the front of the STREAM head read queue.
RPROTDAT	Deliver the control part of a message as data when a process issues a *read*( ).
RPROTDIS	Discard the control part of a message, delivering any data portion, when a process issues a *read*( ).

The *ioctl*( ) function with the I_SRDOPT command shall fail if:

[EINVAL]	The *arg* argument is not valid.

I_GRDOPT Returns the current read mode setting as, described above, in an **int** pointed to by the argument *arg*. Read modes are described in *read*( ).

I_NREAD Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the **int** pointed to by *arg*. The return value for the command shall be the number of messages on the STREAM head read queue. For example, if 0 is returned in *arg*, but the *ioctl*( ) return value is greater than 0, this indicates that a zero-length message is next on the queue.

I_FDINSERT Creates a message from specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The *arg* argument points to a **strfdinsert** structure.

The application shall ensure that the *len* member in the **ctlbuf strbuf** structure is set to the size of a **t_uscalar_t** plus the number of bytes of control information to be sent with the message. The *fildes* member specifies the file descriptor of the other STREAM, and the *offset* member, which must be suitably aligned for use as a **t_uscalar_t**, specifies the offset from the start of the control buffer where I_FDINSERT shall store a **t_uscalar_t** whose interpretation is specific to the STREAM end. The application shall ensure that the *len* member in the **databuf strbuf** structure is set to the number of bytes of data information to be sent with the message, or to 0 if no data part is to be sent.

The *flags* member specifies the type of message to be created. A normal message is created if *flags* is set to 0, and a high-priority message is created if *flags* is set to RS_HIPRI. For non-priority messages, I_FDINSERT shall block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full and O_NONBLOCK is set. Instead, it fails and sets *errno* to [EAGAIN].

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.

The *ioctl*( ) function with the I_FDINSERT command shall fail if:

[EAGAIN] A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.

[EAGAIN] or [ENOSR]
Buffers cannot be allocated for the message that is to be created.

[EINVAL] One of the following:

— The *fildes* member of the **strfdinsert** structure is not a valid, open STREAM file descriptor.

— The size of a **t_uscalar_t** plus *offset* is greater than the *len* member for the buffer specified through **ctlbuf**.

— The *offset* member does not specify a properly-aligned location in the data buffer.

— An undefined value is stored in *flags*.

[ENXIO] Hangup received on the STREAM identified by either the *fildes* argument or the *fildes* member of the **strfdinsert** structure.

[ERANGE] The *len* member for the buffer specified through **databuf** does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module or the *len* member for the buffer specified through **databuf** is larger than the maximum configured size of the data part of a message; or the *len* member for the buffer specified through **ctlbuf** is larger than the maximum configured size of the control part of a message.

I_STR Constructs an internal STREAMS *ioctl*( ) message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send *ioctl*( ) requests to downstream modules and drivers. It allows information to be sent with *ioctl*( ), and returns to the process any information sent upstream by the downstream recipient. I_STR shall block until the system responds with either a positive or negative acknowledgement message, or until the request times out after some period of time. If the request times out, it shall fail with *errno* set to [ETIME].

At most, one I_STR can be active on a STREAM. Further I_STR calls shall block until the active I_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has no effect on this call.

To send requests downstream, the application shall ensure that *arg* points to a **strioctl** structure.

The *ic_cmd* member is the internal *ioctl*( ) command intended for a downstream module or driver and *ic_timout* is the number of seconds (−1=infinite, 0=use implementation-defined timeout interval, >0=as specified) an I_STR request shall wait for acknowledgement before timing out. *ic_len* is the number of bytes in the data argument, and *ic_dp* is a pointer to the data argument. The *ic_len* member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the buffer pointed to by *ic_dp* should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return).

The STREAM head shall convert the information pointed to by the **strioctl** structure to an internal *ioctl*( ) command message and sends it downstream.

The *ioctl*( ) function with the I_STR command shall fail if:

[EAGAIN] or [ENOSR]
Unable to allocate buffers for the *ioctl*( ) message.

[EINVAL] The *ic_len* member is less than 0 or larger than the maximum configured size of the data part of a message, or *ic_timout* is less than −1.

[ENXIO] Hangup received on *fildes*.

[ETIME] A downstream *ioctl*( ) timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hangup is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the *ioctl*( ) command sent downstream fails. For these cases, I_STR shall fail with *errno* set to the value in the message.

I_SWROPT Sets the write mode using the value of the argument *arg*. Valid bit settings for *arg* are:

SNDZERO Send a zero-length message downstream when a *write*( ) of 0 bytes occurs. To not send a zero-length message when a *write*( ) of 0 bytes occurs, the application shall ensure that this bit is not set in *arg* (for example, *arg* would be set to 0).

The *ioctl*( ) function with the I_SWROPT command shall fail if:

[EINVAL] *arg* is not the above value.

I_GWROPT Returns the current write mode setting, as described above, in the **int** that is pointed to by the argument *arg*.

I_SENDFD Creates a new reference to the open file description associated with the file descriptor *arg*, and writes a message on the STREAMS-based pipe *fildes* containing this reference, together with the user ID and group ID of the calling process.

The *ioctl*( ) function with the I_SENDFD command shall fail if:

[EAGAIN] The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queue of the receiving STREAM head is full and cannot accept the message sent by I_SENDFD.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument is not connected to a STREAM pipe.

[ENXIO] Hangup received on *fildes*.

I_RECVFD Retrieves the reference to an open file description from a message written to a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The *arg* argument is a pointer to a **strrecvfd** data structure as defined in **<stropts.h>**.

The *fd* member is a file descriptor. The *uid* and *gid* members are the effective user ID and effective group ID, respectively, of the sending process.

If O_NONBLOCK is not set, I_RECVFD shall block until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD shall fail with *errno* set to [EAGAIN] if no message is present at the STREAM head.

If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor shall be allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the *fd* member of the **strrecvfd** structure pointed to by *arg*.

The *ioctl*( ) function with the I_RECVFD command shall fail if:

[EAGAIN] A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.

[EBADMSG] The message at the STREAM head read queue is not a message containing a passed file descriptor.

[EMFILE] The process has the maximum number of file descriptors currently open that it is allowed.

[ENXIO] Hangup received on *fildes*.

I_LIST Allows the process to list all the module names on the STREAM, up to and including the topmost driver name. If *arg* is a null pointer, the return value shall be the number of modules, including the driver, that are on the STREAM pointed to by *fildes*. This lets the process allocate enough space for the module names. Otherwise, it should point to a **str_list** structure.

The *sl_nmods* member indicates the number of entries the process has allocated in the array. Upon return, the *sl_modlist* member of the **str_list** structure shall contain the list of module names, and the number of entries that have been filled into the *sl_modlist* array is found in the *sl_nmods* member (the number includes the number of modules including the driver). The return value from *ioctl*( ) shall be 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (*sl_nmods*) is satisfied.

The *ioctl*( ) function with the I_LIST command shall fail if:

[EINVAL] The *sl_nmods* member is less than 1.

[EAGAIN] or [ENOSR]
Unable to allocate buffers.

I_ATMARK Allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The *arg* argument determines how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:

ANYMARK	Check if the message is marked.
LASTMARK	Check if the message is the last one marked on the queue.

The bitwise-inclusive OR of the flags ANYMARK and LASTMARK is permitted.

The return value shall be 1 if the mark condition is satisfied; otherwise, the value shall be 0.

The *ioctl*( ) function with the I_ATMARK command shall fail if:

[EINVAL] Invalid *arg* value.

I_CKBAND Checks if the message of a given priority band exists on the STREAM head read queue. This shall return 1 if a message of the given priority exists, 0 if no such message exists, or –1 on error. *arg* should be of type **int**.

The *ioctl*( ) function with the I_CKBAND command shall fail if:

[EINVAL] Invalid *arg* value.

I_GETBAND Returns the priority band of the first message on the STREAM head read queue in the integer referenced by *arg*.

The *ioctl*( ) function with the I_GETBAND command shall fail if:

[ENODATA] No message on the STREAM head read queue.

I_CANPUT Checks if a certain band is writable. *arg* is set to the priority band in question. The return value shall be 0 if the band is flow-controlled, 1 if the band is writable, or –1 on error.

The *ioctl*( ) function with the I_CANPUT command shall fail if:

[EINVAL] Invalid *arg* value.

I_SETCLTIME This request allows the process to set the time the STREAM head shall delay when a STREAM is closing and there is data on the write queues. Before closing each module or driver, if there is data on its write queue, the STREAM head shall delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, it shall be flushed. The *arg* argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, an implementation-defined default timeout interval is used.

The *ioctl*( ) function with the I_SETCLTIME command shall fail if:

[EINVAL] Invalid *arg* value.

I_GETCLTIME Returns the close time delay in the integer pointed to by *arg*.

Multiplexed STREAMS Configurations

The following commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-defined default timeout interval.

I_LINK Connects two STREAMs, where *fildes* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. The STREAM designated by *arg* is connected below the multiplexing driver. I_LINK requires the multiplexing driver to send an acknowledgement message to the STREAM head regarding the connection. This call shall return a multiplexer ID number (an identifier used to disconnect the multiplexer; see I_UNLINK) on success, and −1 on failure.

The *ioctl*( ) function with the I_LINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR] Unable to allocate STREAMS storage to perform the I_LINK.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer; or the specified I_LINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK fails with *errno* set to the value in the message.

I_UNLINK Disconnects the two STREAMs specified by *fildes* and *arg*. *fildes* is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_LINK *ioctl*( ) command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were connected to *fildes* shall be disconnected. As in I_LINK, this command requires acknowledgement.

The *ioctl*( ) function with the I_UNLINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR] Unable to allocate buffers for the acknowledgement message.

[EINVAL] Invalid multiplexer ID number.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK shall fail with *errno* set to the value in the message.

I_PLINK Creates a *persistent connection* between two STREAMs, where *fildes* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. This call shall create a persistent connection which can exist even if the file descriptor *fildes* associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by *arg* gets connected via a persistent connection below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgement message to the STREAM head. This call shall return a multiplexer ID number (an identifier that may be used to disconnect the multiplexer; see I_PUNLINK) on success, and –1 on failure.

The *ioctl*( ) function with the I_PLINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR]
Unable to allocate STREAMS storage to perform the I_PLINK.

[EBADF] The *arg* argument is not a valid, open file descriptor.

[EINVAL] The *fildes* argument does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer; or the specified I_PLINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An I_PLINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PLINK shall fail with *errno* set to the value in the message.

I_PUNLINK Disconnects the two STREAMs specified by *fildes* and *arg* from a persistent connection. The *fildes* argument is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_PLINK *ioctl*( ) command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs which are persistent connections to *fildes* shall be disconnected. As in I_PLINK, this command requires the multiplexing driver to acknowledge the request.

The *ioctl*( ) function with the I_PUNLINK command shall fail if:

[ENXIO] Hangup received on *fildes*.

[ETIME] Timeout before acknowledgement message was received at STREAM head.

[EAGAIN] or [ENOSR]	Unable to allocate buffers for the acknowledgement message.
[EINVAL]	Invalid multiplexer ID number.

An I_PUNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hangup is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PUNLINK shall fail with *errno* set to the value in the message.

RETURN VALUE

Upon successful completion, *ioctl*( ) shall return a value other than −1 that depends upon the STREAMS device control function. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Under the following general conditions, *ioctl*( ) shall fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[EINTR]	A signal was caught during the *ioctl*( ) operation.
[EINVAL]	The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, then *ioctl*( ) shall fail if:

[EINVAL]	The *request* or *arg* argument is not valid for this device.
[EIO]	Some physical I/O error has occurred.
[ENOTTY]	The *fildes* argument is not associated with a STREAMS device that accepts control functions.
[ENXIO]	The *request* and *arg* arguments are valid for this device driver, but the service requested cannot be performed on this particular sub-device.
[ENODEV]	The *fildes* argument refers to a valid STREAMS device, but the corresponding device driver does not support the *ioctl*( ) function.

If a STREAM is connected downstream from a multiplexer, any *ioctl*( ) command except I_UNLINK and I_PUNLINK shall set *errno* to [EINVAL].

EXAMPLES

None.

APPLICATION USAGE

The implementation-defined timeout interval for STREAMS has historically been 15 seconds.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *fcntl*( ), *getmsg*( ), *open*( ), *pipe*( ), *poll*( ), *putmsg*( ), *read*( ), *sigaction*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The Open Group Corrigendum U028/4 is applied, correcting text in the I_FDINSERT, [EINVAL] case to refer to *ctlbuf*.

This function is marked as part of the XSI STREAMS Option Group.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

isalnum — test for an alphanumeric character

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isalnum*( ) function shall test whether *c* is a character of class **alpha** or **digit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale. |

The *c* argument is an **int**, the value of which the application shall ensure is representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isalnum*( ) function shall return non-zero if *c* is an alphanumeric character; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, **<stdio.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

isalpha — test for an alphabetic character

SYNOPSIS

```
#include <ctype.h>

int isalpha(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isalpha*( ) function shall test whether *c* is a character of class **alpha** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *isalpha*( ) function shall return non-zero if *c* is an alphabetic character; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, **<stdio.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

isascii — test for a 7-bit US-ASCII character

SYNOPSIS

XSI

```
#include <ctype.h>

int isascii(int c);
```

DESCRIPTION

The *isascii*( ) function shall test whether *c* is a 7-bit US-ASCII character code.

The *isascii*( ) function is defined on all integer values.

RETURN VALUE

The *isascii*( ) function shall return non-zero if *c* is a 7-bit US-ASCII character code between 0 and octal 0177 inclusive; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

isastream — test a file descriptor (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>

int isastream(int fildes);
```

DESCRIPTION

The *isastream*( ) function shall test whether *fildes*, an open file descriptor, is associated with a STREAMS-based file.

RETURN VALUE

Upon successful completion, *isastream*( ) shall return 1 if *fildes* refers to a STREAMS-based file and 0 if not. Otherwise, *isastream*( ) shall return −1 and set *errno* to indicate the error.

ERRORS

The *isastream*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

isatty — test for a terminal device

SYNOPSIS

```
#include <unistd.h>

int isatty(int fildes);
```

DESCRIPTION

The *isatty*( ) function shall test whether *fildes*, an open file descriptor, is associated with a terminal device.

RETURN VALUE

The *isatty*( ) function shall return 1 if *fildes* is associated with a terminal; otherwise, it shall return 0 and may set *errno* to indicate the error.

ERRORS

The *isatty*( ) function may fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[ENOTTY]	The *fildes* argument is not associated with a terminal.

EXAMPLES

None.

APPLICATION USAGE

The *isatty*( ) function does not necessarily indicate that a human being is available for interaction via *fildes*. It is quite possible that non-terminal devices are connected to the communications line.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional setting of *errno* to indicate an error is added.
- The [EBADF] and [ENOTTY] optional error conditions are added.

NAME

isblank — test for a blank character

SYNOPSIS

```
#include <ctype.h>

int isblank(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isblank*( ) function shall test whether *c* is a character of class **blank** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is a type **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isblank*( ) function shall return non-zero if *c* is a <blank>; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

iscntrl — test for a control character

SYNOPSIS

```
#include <ctype.h>

int iscntrl(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iscntrl*( ) function shall test whether *c* is a character of class **cntrl** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is a type **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iscntrl*( ) function shall return non-zero if *c* is a control character; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

isdigit — test for a decimal digit

SYNOPSIS

```
#include <ctype.h>

int isdigit(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isdigit*( ) function shall test whether *c* is a character of class **digit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isdigit*( ) function shall return non-zero if *c* is a decimal digit; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

isfinite — test for finite value

SYNOPSIS

```
#include <math.h>

int isfinite(real-floating x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isfinite*( ) macro shall determine whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

RETURN VALUE

The *isfinite*( ) macro shall return a non-zero value if and only if its argument has a finite value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fpclassify( ), *isinf*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

isgraph — test for a visible character

SYNOPSIS

```
#include <ctype.h>

int isgraph(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isgraph*( ) function shall test whether *c* is a character of class **graph** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isgraph*( ) function shall return non-zero if *c* is a character with a visible representation; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

isgreater — test if x greater than y

SYNOPSIS

```
#include <math.h>

int isgreater(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isgreater*( ) macro shall determine whether its first argument is greater than its second argument. The value of *isgreater*(x, y) shall be equal to $(x) > (y)$; however, unlike $(x) > (y)$, *isgreater*(x, y) shall not raise the invalid floating-point exception when x and y are unordered.

RETURN VALUE

Upon successful completion, the *isgreater*( ) macro shall return the value of $(x) > (y)$.

If x or y is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreaterequal( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899: 1999 standard.

NAME

isgreaterequal — test if x greater than or equal to y

SYNOPSIS

```
#include <math.h>

int isgreaterequal(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isgreaterequal*( ) macro shall determine whether its first argument is greater than or equal to its second argument. The value of *isgreaterequal*(*x*, *y*) shall be equal to (*x*) >= (*y*); however, unlike (*x*) >= (*y*), *isgreaterequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered.

RETURN VALUE

Upon successful completion, the *isgreaterequal*( ) macro shall return the value of (*x*) >= (*y*).

If *x* or *y* is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreater( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

isinf — test for infinity

SYNOPSIS

```
#include <math.h>

int isinf(real-floating x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isinf*( ) macro shall determine whether its argument value is an infinity (positive or negative). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

RETURN VALUE

The *isinf*( ) macro shall return a non-zero value if and only if its argument has an infinite value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fpclassify( ), *isfinite*( ), *isnan*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

isless — test if x is less than y

SYNOPSIS

```
#include <math.h>

int isless(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isless*( ) macro shall determine whether its first argument is less than its second argument. The value of *isless*(x, y) shall be equal to $(x) < (y)$; however, unlike $(x) < (y)$, *isless*(x, y) shall not raise the invalid floating-point exception when x and y are unordered.

RETURN VALUE

Upon successful completion, the *isless*( ) macro shall return the value of $(x) < (y)$.

If x or y is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreater( ), *isgreaterequal*( ), *islessequal*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

islessequal — test if x is less than or equal to y

SYNOPSIS

```
#include <math.h>

int islessequal(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *islessequal*( ) macro shall determine whether its first argument is less than or equal to its second argument. The value of *islessequal*(*x*, *y*) shall be equal to (*x*) <= (*y*); however, unlike (*x*) <= (*y*), *islessequal*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered.

RETURN VALUE

Upon successful completion, the *islessequal*( ) macro shall return the value of (*x*) <= (*y*).

If *x* or *y* is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreater( ), *isgreaterequal*( ), *isless*( ), *islessgreater*( ), *isunordered*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

islessgreater — test if x is less than or greater than y

SYNOPSIS

```
#include <math.h>

int islessgreater(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *islessgreater*( ) macro shall determine whether its first argument is less than or greater than its second argument. The *islessgreater*(*x*, *y*) macro is similar to (*x*) < (*y*) | | (*x*) > (*y*); however, *islessgreater*(*x*, *y*) shall not raise the invalid floating-point exception when *x* and *y* are unordered (nor shall it evaluate *x* and *y* twice).

RETURN VALUE

Upon successful completion, the *islessgreater*( ) macro shall return the value of (*x*) < (*y*) | | (*x*) > (*y*).

If *x* or *y* is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreater( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *isunordered*( ), the Base Definitions volume of IEEE Std 1003.1-200x **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

islower — test for a lowercase letter

SYNOPSIS

```
#include <ctype.h>

int islower(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *islower*( ) function shall test whether *c* is a character of class **lower** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *islower*( ) function shall return non-zero if *c* is a lowercase letter; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

Testing for a Lowercase Letter

The following example tests whether the value is a lowercase letter, based on the locale of the user, then uses it as part of a key value.

```
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>
...
char *keystr;
int elementlen, len;
char c;
...
setlocale(LC_ALL, "");
...
len = 0;
while (len < elementlen) {
    c = (char) (rand() % 256);
...
    if (islower(c))
        keystr[len++] = c;
    }
...
```

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements. |

An example is added. |

NAME

isnan — test for a NaN

SYNOPSIS

```
#include <math.h>

int isnan(real-floating x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isnan*( ) macro shall determine whether its argument value is a NaN. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

RETURN VALUE

The *isnan*( ) macro shall return a non-zero value if and only if its argument has a NaN value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fpclassify( ), *isfinite*( ), *isinf*( ), *isnormal*( ), *signbit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate the return value when NaN is not supported. This text was previously published in the APPLICATION USAGE section.

Issue 6

Entry re-written for alignment with the ISO/IEC 9899: 1999 standard.

NAME

isnormal — test for a normal value

SYNOPSIS

```
#include <math.h>

int isnormal(real-floating x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isnormal*( ) macro shall determine whether its argument value is normal (neither zero, subnormal, infinite, nor NaN). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

RETURN VALUE

The *isnormal*( ) macro shall return a non-zero value if and only if its argument has a normal value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fpclassify( ), *isfinite*( ), *isinf*( ), *isnan*( ), *signbit*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

isprint — test for a printable character |

SYNOPSIS

```
#include <ctype.h>

int isprint(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isprint*( ) function shall test whether *c* is a character of class **print** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isprint*( ) function shall return non-zero if *c* is a printable character; otherwise, it shall return 0. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

ispunct — test for a punctuation character

SYNOPSIS

```
#include <ctype.h>

int ispunct(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ispunct*( ) function shall test whether *c* is a character of class **punct** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *ispunct*( ) function shall return non-zero if *c* is a punctuation character; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

isspace — test for a white-space character

SYNOPSIS

```
#include <ctype.h>

int isspace(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isspace*( ) function shall test whether *c* is a character of class **space** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isspace*( ) function shall return non-zero if *c* is a white-space character; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isupper*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

isunordered — test if arguments are unordered

SYNOPSIS

```
#include <math.h>

int isunordered(real-floating x, real-floating y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isunordered*( ) macro shall determine whether its arguments are unordered.

RETURN VALUE

Upon successful completion, the *isunordered*( ) macro shall return 1 if its arguments are unordered, and 0 otherwise.

If *x* or *y* is NaN, 0 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The relational and equality operators support the usual mathematical relationships between numeric values. For any ordered pair of numeric values, exactly one of the relationships (less, greater, and equal) is true. Relational operators may raise the invalid floating-point exception when argument values are NaNs. For a NaN and a numeric value, or for two NaNs, just the unordered relationship is true. This macro is a quiet (non-floating-point exception raising) version of a relational operator. It facilitates writing efficient code that accounts for NaNs without suffering the invalid floating-point exception. In the SYNOPSIS section, **real-floating** indicates that the argument shall be an expression of **real-floating** type.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isgreater( ), *isgreaterequal*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

isupper — test for an uppercase letter

SYNOPSIS

```
#include <ctype.h>

int isupper(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isupper*( ) function shall test whether *c* is a character of class **upper** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isupper*( ) function shall return non-zero if *c* is an uppercase letter; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswalnum — test for an alphanumeric wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswalnum(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswalnum*( ) function shall test whether *wc* is a wide-character code representing a character of class **alpha** or **digit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswalnum*( ) function shall return non-zero if *wc* is an alphanumeric wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, **<stdio.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswalpha — test for an alphabetic wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswalpha(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswalpha*( ) function shall test whether *wc* is a wide-character code representing a character of class **alpha** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character | code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

RETURN VALUE

The *iswalpha*( ) function shall return non-zero if *wc* is an alphabetic wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, **<stdio.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswblank — test for a blank wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswblank(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswblank*( ) function shall test whether *wc* is a wide-character code representing a character of class **blank** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswblank*( ) function shall return non-zero if *wc* is a blank wide-character code; otherwise, it shall return 0. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**, **<wctype.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

iswcntrl — test for a control wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswcntrl(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswcntrl*( ) function shall test whether *wc* is a wide-character code representing a character of class **cntrl** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale. |

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswcntrl*( ) function shall return non-zero if *wc* is a control wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswctype — test character for a specified class

SYNOPSIS

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t charclass);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswctype*( ) function shall determine whether the wide-character code *wc* has the character class *charclass*, returning true or false. The *iswctype*( ) function is defined on WEOF and wide-character codes corresponding to the valid character encodings in the current locale. If the *wc* argument is not in the domain of the function, the result is undefined. If the value of *charclass* is invalid (that is, not obtained by a call to *wctype*( ) or *charclass* is invalidated by a subsequent call to *setlocale*( ) that has affected category *LC_CTYPE*) the result is unspecified.

RETURN VALUE

The *iswctype*( ) function shall return non-zero (true) if and only if *wc* has the property described by *charclass*. CX If *charclass* is 0, *iswctype*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

Testing for a Valid Character

```
#include <wctype.h>
...
int yes_or_no;
wint_t wc;
wctype_t valid_class;
...
if ((valid_class=wctype("vowel")) == (wctype_t)0)
    /* Invalid character class. */
yes_or_no=iswctype(wc,valid_class);
```

APPLICATION USAGE

The twelve strings `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"`, and `"xdigit"` are reserved for the standard character classes. In the table below, the functions in the left column are equivalent to the functions in the right column.

```
iswalnum(wc)    iswctype(wc, wctype("alnum"))
iswalpha(wc)    iswctype(wc, wctype("alpha"))
iswblank(wc)    iswctype(wc, wctype("blank"))
iswcntrl(wc)    iswctype(wc, wctype("cntrl"))
iswdigit(wc)    iswctype(wc, wctype("digit"))
iswgraph(wc)    iswctype(wc, wctype("graph"))
iswlower(wc)    iswctype(wc, wctype("lower"))
iswprint(wc)    iswctype(wc, wctype("print"))
iswpunct(wc)    iswctype(wc, wctype("punct"))
iswspace(wc)    iswctype(wc, wctype("space"))
```

```
iswupper(wc)    iswctype(wc, wctype("upper"))
iswxdigit(wc)   iswctype(wc, wctype("xdigit"))
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), *wctype*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

CHANGE HISTORY

First released as World-wide Portability Interfaces in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The behavior of $n=0$ is now described.

An example is added.

A new function, *iswblank*( ), is added to the list in the APPLICATION USAGE.

NAME

iswdigit — test for a decimal digit wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswdigit(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswdigit*( ) function shall test whether *wc* is a wide-character code representing a character of class **digit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswdigit*( ) function shall return non-zero if *wc* is a decimal digit wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term "must" for application requirements.

NAME

iswgraph — test for a visible wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswgraph(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswgraph*( ) function shall test whether *wc* is a wide-character code representing a character of class **graph** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswgraph*( ) function shall return non-zero if *wc* is a wide-character code with a visible representation; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

iswlower — test for a lowercase letter wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswlower(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswlower*( ) function shall test whether *wc* is a wide-character code representing a character of class **lower** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswlower*( ) function shall return non-zero if *wc* is a lowercase letter wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

iswprint — test for a printable wide-character code |

SYNOPSIS

```
#include <wctype.h>

int iswprint(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswprint*( ) function shall test whether *wc* is a wide-character code representing a character of class **print** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswprint*( ) function shall return non-zero if *wc* is a printable wide-character code; otherwise, it shall return 0. |

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswpunct — test for a punctuation wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswpunct(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswpunct*( ) function shall test whether *wc* is a wide-character code representing a character of class **punct** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswpunct*( ) function shall return non-zero if *wc* is a punctuation wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899: 1990/Amendment 1: 1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswspace — test for a white-space wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswspace(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswspace*( ) function shall test whether *wc* is a wide-character code representing a character of class **space** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswspace*( ) function shall return non-zero if *wc* is a white-space wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswupper*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswupper — test for an uppercase letter wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswupper(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswupper*( ) function shall test whether *wc* is a wide-character code representing a character of class **upper** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswupper*( ) function shall return non-zero if *wc* is an uppercase letter wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswxdigit*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

iswxdigit — test for a hexadecimal digit wide-character code

SYNOPSIS

```
#include <wctype.h>

int iswxdigit(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *iswxdigit*( ) function shall test whether *wc* is a wide-character code representing a character of class **xdigit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *wc* argument is a **wint_t**, the value of which the application shall ensure is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro WEOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *iswxdigit*( ) function shall return non-zero if *wc* is a hexadecimal digit wide-character code; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for classification of wide-character codes.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalnum( ), *iswalpha*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *setlocale*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

isxdigit — test for a hexadecimal digit

SYNOPSIS

```
#include <ctype.h>

int isxdigit(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *isxdigit*( ) function shall test whether *c* is a character of class **xdigit** in the program's current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

The *c* argument is an **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the argument has any other value, the behavior is undefined. |

RETURN VALUE

The *isxdigit*( ) function shall return non-zero if *c* is a hexadecimal digit; otherwise, it shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

To ensure applications portability, especially across natural languages, only this function and those listed in the SEE ALSO section should be used for character classification.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalnum( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

j0, j1, jn — Bessel functions of the first kind

SYNOPSIS

XSI
```
#include <math.h>

double j0(double x);
double j1(double x);
double jn(int n, double x);
```

DESCRIPTION

The *j0*( ), *j1*( ), and *jn*( ) functions shall compute Bessel functions of *x* of the first kind of orders 0, 1, and *n*, respectively.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the relevant Bessel value of *x* of the first kind.

If the *x* argument is too large in magnitude, or the correct result would cause underflow, 0 shall be returned and a range error may occur.

If *x* is NaN, a NaN shall be returned.

ERRORS

These functions may fail if:

Range Error The value of *x* was too large in magnitude, or an underflow occurred.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised. |

No other errors shall occur.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *y0*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The may fail [EDOM] error is removed for the case for NaN.

The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling with the ISO / IEC 9899: 1999 standard. |

NAME

jrand48 — generate a uniformly distributed pseudo-random long signed integer

SYNOPSIS

XSI

```
#include <stdlib.h>

long jrand48(unsigned short xsubi[3]);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

kill — send a signal to a process or a group of processes

SYNOPSIS

CX
```
#include <signal.h>

int kill(pid_t pid, int sig);
```

DESCRIPTION

The *kill*( ) function shall send a signal to a process or a group of processes specified by *pid*. The signal to be sent is specified by *sig* and is either one from the list given in **<signal.h>** or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

For a process to have permission to send a signal to a process designated by *pid*, unless the sending process has appropriate privileges, the real or effective user ID of the sending process shall match the real or saved set-user-ID of the receiving process.

If *pid* is greater than 0, *sig* shall be sent to the process whose process ID is equal to *pid*.

If *pid* is 0, *sig* shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If *pid* is −1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

If *pid* is negative, but not −1, *sig* shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of *pid*, and for which the process has permission to send a signal.

If the value of *pid* causes *sig* to be generated for the sending process, and if *sig* is not blocked for the calling thread and if no other thread has *sig* unblocked or is waiting in a *sigwait*( ) function for *sig*, either *sig* or at least one pending unblocked signal shall be delivered to the sending thread before *kill*( ) returns.

The user ID tests described above shall not be applied when sending SIGCONT to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals, including the null signal. In particular, the system may deny the existence of some or all of the processes specified by *pid*.

The *kill*( ) function is successful if the process has permission to send *sig* to any of the processes specified by *pid*. If *kill*( ) fails, no signal shall be sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *kill*( ) function shall fail if:

[EINVAL]	The value of the *sig* argument is an invalid or unsupported signal number.
[EPERM]	The process does not have permission to send the signal to any receiving process.
[ESRCH]	No process or process group can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The semantics for permission checking for *kill*( ) differed between System V and most other implementations, such as Version 7 or 4.3 BSD. The semantics chosen for this volume of IEEE Std 1003.1-200x agree with System V. Specifically, a set-user-ID process cannot protect itself against signals (or at least not against SIGKILL) unless it changes its real user ID. This choice allows the user who starts an application to send it signals even if it changes its effective user ID. The other semantics give more power to an application that wants to protect itself from the user who ran it.

Some implementations provide semantic extensions to the *kill*( ) function when the absolute value of *pid* is greater than some maximum, or otherwise special, value. Negative values are a flag to *kill*( ). Since most implementations return [ESRCH] in this case, this behavior is not included in this volume of IEEE Std 1003.1-200x, although a conforming implementation could provide such an extension.

The implementation-defined processes to which a signal cannot be sent may include the scheduler or *init*.

There was initially strong sentiment to specify that, if *pid* specifies that a signal be sent to the calling process and that signal is not blocked, that signal would be delivered before *kill*( ) returns. This would permit a process to call *kill*( ) and be guaranteed that the call never return. However, historical implementations that provide only the *signal*( ) function make only the weaker guarantee in this volume of IEEE Std 1003.1-200x, because they only deliver one signal each time a process enters the kernel. Modifications to such implementations to support the *sigaction*( ) function generally require entry to the kernel following return from a signal-catching function, in order to restore the signal mask. Such modifications have the effect of satisfying the stronger requirement, at least when *sigaction*( ) is used, but not necessarily when *signal*( ) is used. The developers of this volume of IEEE Std 1003.1-200x considered making the stronger requirement except when *signal*( ) is used, but felt this would be unnecessarily complex. Implementors are encouraged to meet the stronger requirement whenever possible. In practice, the weaker requirement is the same, except in the rare case when two signals arrive during a very short window. This reasoning also applies to a similar requirement for *sigprocmask*( ).

In 4.2 BSD, the SIGCONT signal can be sent to any descendant process regardless of user-ID security checks. This allows a job control shell to continue a job even if processes in the job have altered their user IDs (as in the *su* command). In keeping with the addition of the concept of sessions, similar functionality is provided by allowing the SIGCONT signal to be sent to any process in the same session regardless of user ID security checks. This is less restrictive than BSD in the sense that ancestor processes (in the same session) can now be the recipient. It is more restrictive than BSD in the sense that descendant processes that form new sessions are now subject to the user ID checks. A similar relaxation of security is not necessary for the other job control signals since those signals are typically sent by the terminal driver in recognition of special characters being typed; the terminal driver bypasses all security checks.

In secure implementations, a process may be restricted from sending a signal to a process having a different security label. In order to prevent the existence or nonexistence of a process from being used as a covert channel, such processes should appear nonexistent to the sender; that is, [ESRCH] should be returned, rather than [EPERM], if *pid* refers only to such processes.

Existing implementations vary on the result of a *kill*( ) with *pid* indicating an inactive process (a terminated process that has not been waited for by its parent). Some indicate success on such a call (subject to permission checking), while others give an error of [ESRCH]. Since the definition of process lifetime in this volume of IEEE Std 1003.1-200x covers inactive processes, the [ESRCH] error as described is inappropriate in this case. In particular, this means that an application cannot have a parent process check for termination of a particular child with *kill*( ). (Usually this is done with the null signal; this can be done reliably with *waitpid*( ).)

There is some belief that the name *kill*( ) is misleading, since the function is not always intended to cause process termination. However, the name is common to all historical implementations, and any change would be in conflict with the goal of minimal changes to existing application code.

FUTURE DIRECTIONS

None.

SEE ALSO

getpid( ), *raise*( ), *setsid*( ), *sigaction*( ), *sigqueue*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with POSIX Threads Extension.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the second paragraph is reworded to indicate that the saved set-user-ID of the calling process is checked in place of its effective user ID. This is a FIPS requirement.
- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The behavior when *pid* is −1 is now specified. It was previously explicitly unspecified in the POSIX.1-1988 standard.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

killpg — send a signal to a process group

SYNOPSIS

XSI

```
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

DESCRIPTION

The *killpg*( ) function shall send the signal specified by *sig* to the process group specified by *pgrp*.

If *pgrp* is greater than 1, *killpg*(*pgrp*, *sig*) shall be equivalent to *kill*(−*pgrp*, *sig*). If *pgrp* is less than or equal to 1, the behavior of *killpg*( ) is undefined.

RETURN VALUE

Refer to *kill*( ).

ERRORS

Refer to *kill*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpgid( ), *getpid*( ), *kill*( ), *raise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

l64a — convert a 32-bit integer to a radix-64 ASCII string

SYNOPSIS

XSI

```
#include <stdlib.h>

char *l64a(long value);
```

DESCRIPTION

Refer to *a64l*( ).

NAME

labs, llabs — return a long integer absolute value

SYNOPSIS

```
#include <stdlib.h>

long labs(long i);
long long llabs(long long i);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *labs*( ) function shall compute the absolute value of the **long** integer operand *i*. The *llabs*( ) function shall compute the absolute value of the **long long** integer operand *i*. If the result cannot be represented, the behavior is undefined.

RETURN VALUE

The *labs*( ) function shall return the absolute value of the **long** integer operand. The *labs*( ) function shall return the absolute value of the **long long** integer operand.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

Issue 6

The *llabs*( ) function is added for alignment with the ISO/IEC 9899:1999 standard.

NAME

lchown — change the owner and group of a symbolic link

SYNOPSIS

XSI
```
#include <unistd.h>

int lchown(const char *path, uid_t owner, gid_t group);
```

DESCRIPTION

The *lchown*( ) function shall be equivalent to *chown*( ), except in the case where the named file is a symbolic link. In this case, *lchown*( ) shall change the ownership of the symbolic link file itself, while *chown*( ) changes the ownership of the file or directory to which the symbolic link refers. |

RETURN VALUE

Upon successful completion, *lchown*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate an error.

ERRORS

The *lchown*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix of *path*.
[EINVAL]	The owner or group ID is not a value supported by the implementation.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EOPNOTSUPP]	The *path* argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.
[EPERM]	The effective user ID does not match the owner of the file and the process does not have appropriate privileges.
[EROFS]	The file resides on a read-only file system.

The *lchown*( ) function may fail if:

[EIO]	An I/O error occurred while reading or writing to the file system.
[EINTR]	A signal was caught during execution of the function.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

EXAMPLES

Changing the Current Owner of a File

The following example shows how to change the ownership of the symbolic link named **/modules/pass1** to the user ID associated with ‘‘jones’’ and the group ID associated with ‘‘cnd’’.

The numeric value for the user ID is obtained by using the *getpwnam*( ) function. The numeric value for the group ID is obtained by using the *getgrnam*( ) function.

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>

struct passwd *pwd;
struct group  *grp;
char          *path = "/modules/pass1";
...
pwd = getpwnam("jones");
grp = getgrnam("cnd");
lchown(path, pwd->pw_uid, grp->gr_gid);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X / OPEN UNIX extension to BASE.

Issue 6

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

lcong48 — seed a uniformly distributed pseudo-random signed long integer generator

SYNOPSIS

XSI
```
#include <stdlib.h>

void lcong48(unsigned short param[7]);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

ldexp, ldexpf, ldexpl — load exponent of a floating-point number

SYNOPSIS

```
#include <math.h>

double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the quantity $x * 2^{exp}$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return *x* multiplied by 2, raised to the power *exp*.

If these functions would cause overflow, a range error shall occur and *ldexp*( ), *ldexpf*( ), and *ldexpl*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (according to the sign of *x*), respectively.

MX If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0 or ±Inf, *x* shall be returned.

If *exp* is 0, *x* shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

Range Error	The result overflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *frexp*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *ldexpf*( ) and *ldexpl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

ldiv, lldiv — compute quotient and remainder of a long division

SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the **long** integer (for the *ldiv*( ) function) or **long long** integer (for the *lldiv*( ) function) of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, *quot* * *denom*+*rem* shall equal *numer*.

RETURN VALUE

The *ldiv*( ) function shall return a structure of type **ldiv_t**, comprising both the quotient and the remainder. The structure shall include the following members, in any order: |

```
long   quot;    /* Quotient */
long   rem;     /* Remainder */
```

The *lldiv*( ) function shall return a structure of type **lldiv_t**, comprising both the quotient and the remainder. The structure shall include the following members, in any order: |

```
long long   quot;    /* Quotient */
long long   rem;     /* Remainder */
```

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

div( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

Issue 6

The *lldiv*( ) function is added for alignment with the ISO/IEC 9899:1999 standard.

NAME

lfind — find entry in a linear search table

SYNOPSIS

XSI

```
#include <search.h>

void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));

```

DESCRIPTION

Refer to *lsearch*( ).

NAME

lgamma, lgammaf, lgammal — log gamma function

SYNOPSIS

```
#include <math.h>

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
extern int signgam;
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute $\log_e|\Gamma(x)|$ where $\Gamma(x)$ is defined as $\int_0^\infty e^{-t}t^{x-1}dt$. The argument x need not be a non-positive integer ($\Gamma(x)$ is defined over the reals, except the non-positive integers).

XSI The sign of $\Gamma(x)$ is returned in the external integer *signgam*.

CX These functions need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the logarithmic gamma of x.

If x is a non-positive integer, a pole error shall occur and *lgamma*( ), *lgammaf*( ), and *lgammal*( ) shall return +HUGE_VAL, +HUGE_VALF, and +HUGE_VALL, respectively.

If the correct value would cause overflow, a range error shall occur and *lgamma*( ), *lgammaf*( ), and *lgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (having the same sign as the correct value), respectively.

MX If x is NaN, a NaN shall be returned.

If x is 1 or 2, +0 shall be returned.

If x is ±Inf, +Inf shall be returned

ERRORS

These functions shall fail if:

Pole Error	The x argument is a negative integer or zero. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.
Range Error	The result overflows

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *lgamma*( ) function is no longer marked as an extension.

The *lgammaf*( ) and *lgammal*( ) functions are added for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO / IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO / IEC 9899: 1999 standard are marked.

XSI extensions are marked.

NAME

link — link to a file

SYNOPSIS

```
#include <unistd.h>

int link(const char *path1, const char *path2);
```

DESCRIPTION

The *link*( ) function shall create a new link (directory entry) for the existing file, *path1*.

The *path1* argument points to a pathname naming an existing file. The *path2* argument points to a pathname naming the new directory entry to be created. The *link*( ) function shall atomically create a new link for the existing file and the link count of the file shall be incremented by one.

If *path1* names a directory, *link*( ) shall fail unless the process has appropriate privileges and the implementation supports using *link*( ) on directories.

Upon successful completion, *link*( ) shall mark for update the *st_ctime* field of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

If *link*( ) fails, no link shall be created and the link count of the file shall remain unchanged.

The implementation may require that the calling process has permission to access the existing file.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *link*( ) function shall fail if:

[EACCES]	A component of either path prefix denies search permission, or the requested link requires writing in a directory that denies write permission, or the calling process does not have permission to access the existing file and this is required by the implementation.
[EEXIST]	The *path2* argument resolves to an existing file or refers to a symbolic link.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path1* or *path2* argument.
[EMLINK]	The number of links to the file named by *path1* would exceed {LINK_MAX}.
[ENAMETOOLONG]	The length of the *path1* or *path2* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of either path prefix does not exist; the file named by *path1* does not exist; or *path1* or *path2* points to an empty string.
[ENOSPC]	The directory to contain the link cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory.
[EPERM]	The file named by *path1* is a directory and either the calling process does not have appropriate privileges or the implementation prohibits using *link*( ) on directories.

[EROFS] The requested link requires writing in a directory on a read-only file system.

[EXDEV] The link named by *path2* and the file named by *path1* are on different file systems and the implementation does not support links between file systems, XSR or *path1* refers to a named STREAM.

The *link*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path1* or *path2* argument.

[ENAMETOOLONG] As a result of encountering a symbolic link in resolution of the *path1* or *path2* argument, the length of the substituted pathname string exceeded {PATH_MAX}. |

EXAMPLES

Creating a Link to a File

The following example shows how to create a link to a file named **/home/cnd/mod1** by creating a new directory entry named **/modules/pass1**.

```
#include <unistd.h>

char *path1 = "/home/cnd/mod1";
char *path2 = "/modules/pass1";
int   status;
...
status = link (path1, path2);
```

Creating a Link to a File Within a Program

In the following program example, the *link*( ) function links the **/etc/passwd** file (defined as **PASSWDFILE**) to a file named **/etc/opasswd** (defined as **SAVEFILE**), which is used to save the current password file. Then, after removing the current password file (defined as **PASSWDFILE**), the new password file is saved as the current password file using the *link*( ) function again. |

```
#include <unistd.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* Save current password file */
link (PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink (PASSWDFILE);

/* Save new password file as current password file. */
link (LOCKFILE,PASSWDFILE);
```

APPLICATION USAGE

Some implementations do allow links between file systems.

RATIONALE

Linking to a directory is restricted to the superuser in most historical implementations because this capability may produce loops in the file hierarchy or otherwise corrupt the file system. This volume of IEEE Std 1003.1-200x continues that philosophy by prohibiting *link*( ) and *unlink*( ) from doing this. Other functions could do it if the implementor designed such an extension.

Some historical implementations allow linking of files on different file systems. Wording was added to explicitly allow this optional behavior. |

The exception for cross-file system links is intended to apply only to links that are programmatically indistinguishable from ‘‘hard’’ links.

FUTURE DIRECTIONS

None.

SEE ALSO

symlink( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- An explanation is added of action when *path2* refers to a symbolic link.
- The [ELOOP] optional error condition is added.

NAME

lio_listio — list directed I/O (**REALTIME**)

SYNOPSIS

AIO

```
#include <aio.h>

int lio_listio(int mode, struct aiocb *restrict const list[restrict],
    int nent, struct sigevent *restrict sig);

```

DESCRIPTION

The *lio_listio*( ) function shall initiate a list of I/O requests with a single function call.

The *mode* argument takes one of the values LIO_WAIT or LIO_NOWAIT declared in **<aio.h>** and determines whether the function returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* argument is LIO_WAIT, the function shall wait until all I/O is complete and the *sig* argument shall be ignored.

If the *mode* argument is LIO_NOWAIT, the function shall return immediately, and asynchronous notification shall occur, according to the *sig* argument, when all the I/O operations complete. If *sig* is NULL, then no asynchronous notification shall occur. If *sig* is not NULL, asynchronous notification occurs as specified in Section 2.4.1 (on page 478) when all the requests in *list* have completed.

The I/O requests enumerated by *list* are submitted in an unspecified order.

The *list* argument is an array of pointers to **aiocb** structures. The array contains *nent* elements. The array may contain NULL elements, which shall be ignored.

The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in **<aio.h>**. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode* element is equal to LIO_READ, then an I/O operation is submitted as if by a call to *aio_read*( ) with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal to LIO_WRITE, then an I/O operation is submitted as if by a call to *aio_write*( ) with the *aiocbp* equal to the address of the **aiocb** structure.

The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.

The *aio_nbytes* member specifies the number of bytes of data to be transferred.

The members of the **aiocb** structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding **aiocb** structure when used by the *aio_read*( ) and *aio_write*( ) functions.

The *nent* argument specifies how many elements are members of the list; that is, the length of the array.

The behavior of this function is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with *aio_fildes*.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *aiocbp*->*aio_fildes*.

RETURN VALUE

If the *mode* argument has the value LIO_NOWAIT, the *lio_listio*( ) function shall return the value zero if the I/O operations are successfully queued; otherwise, the function shall return the value −1 and set *errno* to indicate the error.

If the *mode* argument has the value LIO_WAIT, the *lio_listio*( ) function shall return the value zero when all the indicated I/O has completed successfully. Otherwise, *lio_listio*( ) shall return a value of −1 and set *errno* to indicate the error.

In either case, the return value only indicates the success or failure of the *lio_listio*( ) call itself, not the status of the individual I/O requests. In some cases one or more of the I/O requests contained in the list may fail. Failure of an individual request does not prevent completion of any other individual request. To determine the outcome of each I/O request, the application shall examine the error status associated with each **aiocb** control block. The error statuses so returned are identical to those returned as the result of an *aio_read*( ) or *aio_write*( ) function.

ERRORS

The *lio_listio*( ) function shall fail if:

[EAGAIN]	The resources necessary to queue all the I/O requests were not available. The application may check the error status for each **aiocb** to determine the individual request(s) that failed.
[EAGAIN]	The number of entries indicated by *nent* would cause the system-wide limit {AIO_MAX} to be exceeded.
[EINVAL]	The *mode* argument is not a proper value, or the value of *nent* was greater than {AIO_LISTIO_MAX}.
[EINTR]	A signal was delivered while waiting for all I/O requests to complete during an LIO_WAIT operation. Note that, since each I/O operation invoked by *lio_listio*( ) may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application shall examine each list element to determine whether the request was initiated, canceled, or completed.
[EIO]	One or more of the individual I/O operations failed. The application may check the error status for each **aiocb** structure to determine the individual request(s) that failed.

In addition to the errors returned by the *lio_listio*( ) function, if the *lio_listio*( ) function succeeds or fails with errors of [EAGAIN], [EINTR], or [EIO], then some of the I/O specified by the list may have been initiated. If the *lio_listio*( ) function fails with an error code other than [EAGAIN], [EINTR], or [EIO], no operations from the list shall have been initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each **aiocb** control block contains the associated error code. The error codes that can be set are the same as would be set by a *read*( ) or *write*( ) function, with the following additional error codes possible:

[EAGAIN]	The requested I/O operation was not queued due to resource limitations.
[ECANCELED]	The requested I/O was canceled before the I/O completed due to an explicit *aio_cancel*( ) request.
[EFBIG]	The *aiocbp*->*aio_lio_opcode* is LIO_WRITE, the file is a regular file, *aiocbp*->*aio_nbytes* is greater than 0, and the *aiocbp*->*aio_offset* is greater than or equal to the offset maximum in the open file description associated with *aiocbp*-

>aio_fildes.

[EINPROGRESS] The requested I/O is in progress.

[EOVERFLOW] The *aiocbp->aio_lio_opcode* is LIO_READ, the file is a regular file, *aiocbp->aio_nbytes* is greater than 0, and the *aiocbp->aio_offset* is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aiocbp->aio_fildes*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Although it may appear that there are inconsistencies in the specified circumstances for error codes, the [EIO] error condition applies when any circumstance relating to an individual operation makes that operation fail. This might be due to a badly formulated request (for example, the *aio_lio_opcode* field is invalid, and *aio_error*( ) returns [EINVAL]) or might arise from application behavior (for example, the file descriptor is closed before the operation is initiated, and *aio_error*( ) returns [EBADF]).

The limitation on the set of error codes returned when operations from the list shall have been initiated enables applications to know when operations have been started and whether *aio_error*( ) is valid for a specific operation.

FUTURE DIRECTIONS

None.

SEE ALSO

aio_read( ), *aio_write*( ), *aio_error*( ), *aio_return*( ), *aio_cancel*( ), *close*( ), *exec*, *exit*( ), *fork*( ), *lseek*( ), *read*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<aio.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Asynchronous Input and Output option.

The *lio_listio*( ) function is marked as part of the Asynchronous Input and Output option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text is added to indicate that for regular files no data transfer occurs past the offset maximum established in the open file description associated with *aiocbp->aio_fildes*. This change is to support large files.
- The [EBIG] and [EOVERFLOW] error conditions are defined. This change is to support large files.

The DESCRIPTION is updated to avoid use of the term “must” for application requirements.

The **restrict** keyword is added to the *lio_listio*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

listen — listen for socket connections and limit the queue of incoming connections

SYNOPSIS

```
#include <sys/socket.h>

int listen(int socket, int backlog);
```

DESCRIPTION

The *listen*( ) function shall mark a connection-mode socket, specified by the *socket* argument, as accepting connections.

The *backlog* argument provides a hint to the implementation which the implementation shall use to limit the number of outstanding connections in the socket's listen queue. Implementations may impose a limit on *backlog* and silently reduce the specified value. Normally, a larger *backlog* argument value shall result in a larger or equal length of the listen queue. Implementations shall support values of *backlog* up to SOMAXCONN, defined in **<sys/socket.h>**.

The implementation may include incomplete connections in its listen queue. The limits on the number of incomplete connections and completed connections queued may be different.

The implementation may have an upper limit on the length of the listen queue—either global or per accepting socket. If *backlog* exceeds this limit, the length of the listen queue is set to the limit.

If *listen*( ) is called with a *backlog* argument value that is less than 0, the function behaves as if it had been called with a *backlog* argument value of 0.

A *backlog* argument of 0 may allow the socket to accept connections, in which case the length of the listen queue may be set to an implementation-defined minimum value.

The socket in use may require the process to have appropriate privileges to use the *listen*( ) function.

RETURN VALUE

Upon successful completions, *listen*( ) shall return 0; otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *listen*( ) function shall fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[EDESTADDRREQ]	The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.
[EINVAL]	The *socket* is already connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The socket protocol does not support *listen*( ).

The *listen*( ) function may fail if:

[EACCES]	The calling process does not have the appropriate privileges.
[EINVAL]	The *socket* has been shut down.
[ENOBUFS]	Insufficient resources are available in the system to complete the call.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *connect*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The DESCRIPTION is updated to describe the relationship of SOMAXCONN and the *backlog* argument.

NAME

llabs — return a long integer absolute value

SYNOPSIS

```
#include <stdlib.h>

long long llabs(long long i);
```

DESCRIPTION

Refer to *labs*( ).

NAME

lldiv — compute quotient and remainder of a long division

SYNOPSIS

```
#include <stdlib.h>

lldiv_t lldiv(long long numer, long long denom);
```

DESCRIPTION

Refer to *ldiv*( ).

NAME

llrint, llrintf, llrintl, — round to nearest integer value using current rounding direction

SYNOPSIS

```
#include <math.h>

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long long**, a domain error shall occur and an unspecified value is returned. |

If the correct value is negative and too large to represent as a **long long**, a domain error shall occur and an unspecified value is returned. |

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions provide floating-to-integer conversions. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the invalid floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the inexact

floating-point exception.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *lrint*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

llround, llroundf, llroundl, — round to nearest integer value

SYNOPSIS

```
#include <math.h>

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long long**, a domain error shall occur and an unspecified value is returned.

If the correct value is negative and too large to represent as a **long long**, a domain error shall occur and an unspecified value is returned.

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions provide floating-to-integer conversions. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the invalid floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the inexact

floating-point exception.

These functions differ from the *llrint*( ) functions in that the default rounding direction for the *llround*( ) functions round halfway cases away from zero and need not raise the inexact floating-point exception for non-integer arguments that round to within the range of the return type.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *lround*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

localeconv — return locale-specific information

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *localeconv*( ) function shall set the components of an object with the type **struct lconv** with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The members of the structure with type **char *** are pointers to strings, any of which (except **decimal_point**) can point to `""`, to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are non-negative numbers, any of which can be {CHAR_MAX} to indicate that the value is not available in the current locale.

The members include the following:

char *decimal_point
The radix character used to format non-monetary quantities.

char *thousands_sep
The character used to separate groups of digits before the decimal-point character in formatted non-monetary quantities.

char *grouping
A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted non-monetary quantities.

char *int_curr_symbol
The international currency symbol applicable to the current locale. The first three characters contain the alphabetic international currency symbol in accordance with those specified in the ISO 4217: 1995 standard. The fourth character (immediately preceding the null byte) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
The local currency symbol applicable to the current locale.

char *mon_decimal_point
The radix character used to format monetary quantities.

char *mon_thousands_sep
The separator for groups of digits before the decimal-point in formatted monetary quantities.

char *mon_grouping
A string whose elements taken as one-byte integer values indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
The string used to indicate a non-negative valued formatted monetary quantity.

char *negative_sign
The string used to indicate a negative valued formatted monetary quantity.

char int_frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in an internationally formatted monetary quantity.

char frac_digits
The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

char p_cs_precedes
Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a non-negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char p_sep_by_space
Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a non-negative formatted monetary quantity. Set to 1 if a space separates the symbol from the XSI value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

char n_cs_precedes
Set to 1 if the **currency_symbol** or **int_curr_symbol** precedes the value for a negative formatted monetary quantity. Set to 0 if the symbol succeeds the value.

char n_sep_by_space
Set to 0 if no space separates the **currency_symbol** or **int_curr_symbol** from the value for a negative formatted monetary quantity. Set to 1 if a space separates the symbol from the XSI value; and set to 2 if a space separates the symbol and the sign string, if adjacent.

char p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a non-negative formatted monetary quantity.

char n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

char int_p_cs_precedes
Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a non-negative internationally formatted monetary quantity. |

char int_n_cs_precedes
Set to 1 or 0 if the **int_curr_symbol** respectively precedes or succeeds the value for a negative internationally formatted monetary quantity. |

char int_p_sep_by_space
Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the value for a non-negative internationally formatted monetary quantity. |

char int_n_sep_by_space
Set to a value indicating the separation of the **int_curr_symbol**, the sign string, and the value for a negative internationally formatted monetary quantity. |

char int_p_sign_posn
Set to a value indicating the positioning of the **positive_sign** for a non-negative internationally formatted monetary quantity.

char int_n_sign_posn
Set to a value indicating the positioning of the **negative_sign** for a negative internationally

formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

{CHAR_MAX}	No further grouping is to be performed.
0	The previous element is to be repeatedly used for the remainder of the digits.
other	The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The values of **p_sep_by_space**, **n_sep_by_space**, **int_p_sep_by_space**, and **int_n_sep_by_space** are interpreted according to the following:

0 No space separates the currency symbol and value.

1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

For **int_p_sep_by_space** and **int_n_sep_by_space**, the fourth character of **int_curr_symbol** is used instead of a space.

The values of **p_sign_posn**, **n_sign_posn**, **int_p_sign_posn**, and **int_n_sign_posn** are interpreted according to the following:

0 Parentheses surround the quantity and **currency_symbol** or **int_curr_symbol**.

1 The sign string precedes the quantity and **currency_symbol** or **int_curr_symbol**.

2 The sign string succeeds the quantity and **currency_symbol** or **int_curr_symbol**.

3 The sign string immediately precedes the **currency_symbol** or **int_curr_symbol**.

4 The sign string immediately succeeds the **currency_symbol** or **int_curr_symbol**.

The implementation shall behave as if no function in this volume of IEEE Std 1003.1-200x calls *localeconv*( ).

CX The *localeconv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *localeconv*( ) function shall return a pointer to the filled-in object. The application shall not modify the structure pointed to by the return value which may be overwritten by a subsequent call to *localeconv*( ). In addition, calls to *setlocale*( ) with the categories *LC_ALL*, *LC_MONETARY*, or *LC_NUMERIC* may overwrite the contents of the structure.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following table illustrates the rules which may be used by four countries to format monetary quantities.

Country	Positive Format	Negative Format	International Format
Italy	L.1.230	–L.1.230	ITL.1.230
Netherlands	F 1.234,56	F –1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56–	NOK 1.234,56
Switzerland	SFrs.1,234.56	SFrs.1,234.56C	CHF 1,234.56

For these four countries, the respective values for the monetary members of the structure returned by *localeconv*( ) are:

	Italy	Netherlands	Norway	Switzerland
int_curr_symbol	"ITL."	"NLG "	"NOK "	"CHF "
currency_symbol	"L."	"F"	"kr"	"SFrs."
mon_decimal_point	""	","	","	"."
mon_thousands_sep	"."	"."	"."	","
mon_grouping	"\3"	"\3"	"\3"	"\3"
positive_sign	""	""	""	""
negative_sign	"-"	"-"	"-"	"C"
int_frac_digits	0	2	2	2
frac_digits	0	2	2	2
p_cs_precedes	1	1	1	1
p_sep_by_space	0	1	0	0
n_cs_precedes	1	1	1	1
n_sep_by_space	0	1	0	0
p_sign_posn	1	1	1	1
n_sign_posn	1	4	2	2
int_p_cs_precedes	1	1	1	1
int_n_cs_precedes	1	1	1	1
int_p_sep_by_space	0	0	0	0
int_n_sep_by_space	0	0	0	0
int_p_sign_posn	1	1	1	1
int_n_sign_posn	1	4	4	2

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *isascii*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcat*( ), *strchr*( ), *strcmp*( ), *strcoll*( ), *strcpy*( ), *strftime*( ), *strlen*( ), *strpbrk*( ), *strspn*( ), *strtok*( ), *strxfrm*( ), *strtod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

The RETURN VALUE section is rewritten to avoid use of the term ‘‘must’’.

This reference page is updated for alignment with the ISO/IEC 9899:1999 standard. |

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

localtime, localtime_r — convert a time value to a broken-down local time

SYNOPSIS

```
#include <time.h>

struct tm *localtime(const time_t *timer);
struct tm *localtime_r(const time_t *restrict timer,
    struct tm *restrict result);

```

DESCRIPTION

For *localtime*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *localtime*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into a broken-down time, expressed as a local time. The function corrects for the timezone and any seasonal time adjustments. Local timezone information is used as though *localtime*( ) calls *tzset*( ).

The relationship between a time in seconds since the Epoch used as an argument to *localtime*( ) and the **tm** structure (defined in the **<time.h>** header) is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time adjustments, where the names in the structure and in the expression correspond.

The same relationship shall apply for *localtime_r*( ).

The *localtime*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The *asctime*( ), *ctime*( ), *gmtime*( ), and *localtime*( ) functions shall return values in one of two static objects: a broken-down time structure and an array of type **char**. Execution of any of the functions may overwrite the information returned in either of these objects by any of the other functions.

The *localtime_r*( ) function shall convert the time in seconds since the Epoch pointed to by *timer* into a broken-down time stored in the structure to which *result* points. The *localtime_r*( ) function shall also return a pointer to that same structure.

Unlike *localtime*( ), the reentrant version is not required to set *tzname*.

RETURN VALUE

The *localtime*( ) function shall return a pointer to the broken-down time structure.

Upon successful completion, *localtime_r*( ) shall return a pointer to the structure pointed to by the argument *result*.

ERRORS

No errors are defined.

EXAMPLES

Getting the Local Date and Time

The following example uses the *time*( ) function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC (the Epoch), *localtime*( ) to convert that value to a broken-down time, and *asctime*( ) to convert the broken-down time values into a printable string.

```
#include <stdio.h>
#include <time.h>

main()
{
    time_t result;

    result = time(NULL);
    printf("%s%ld secs since the Epoch\n",
        asctime(localtime(&result)),
            (long)result);
    return(0);
}
```

This example writes the current time to *stdout* in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

Getting the Modification Time for a File

The following example gets the modification time for a file. The *localtime*( ) function converts the **time_t** value of the last modification date, obtained by a previous call to *stat*( ), into a **tm** structure that contains the year, month, day, and so on.

```
#include <time.h>
...
struct stat statbuf;
...
tm = localtime(&statbuf.st_mtime);
...
```

Timing an Event

The following example gets the current time, converts it to a string using *localtime*( ) and *asctime*( ), and prints it to standard output using *fputs*( ). It then prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
```

```
        minutes_to_event);
    ...
```

APPLICATION USAGE

The *localtime_r*( ) function is thread-safe and returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *getdate*( ), *gmtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A note indicating that the *localtime*( ) function need not be reentrant is added to the DESCRIPTION.

The *localtime_r*( ) function is included for alignment with the POSIX Threads Extension.

Issue 6

The *localtime_r*( ) function is marked as part of the Thread-Safe Functions option.

Extensions beyond the ISO C standard are now marked.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *localtime_r*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

Examples are added.

NAME

lockf — record locking on files

SYNOPSIS

XSI

```
#include <unistd.h>

int lockf(int fildes, int function, off_t size);
```

DESCRIPTION

The *lockf*( ) function shall lock sections of a file with advisory-mode locks. Calls to *lockf*( ) from other threads which attempt to lock the locked file section shall either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates. Record locking with *lockf*( ) shall be supported for regular files and may be supported for other files.

The *fildes* argument is an open file descriptor. To establish a lock with this function, the file descriptor shall be opened with write-only permission (O_WRONLY) or with read/write permission (O_RDWR).

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are defined in **<unistd.h>** as follows:

Function	Description
F_ULOCK	Unlock locked sections.
F_LOCK	Lock a section for exclusive use.
F_TLOCK	Test and lock a section for exclusive use.
F_TEST	Test a section for locks by other processes.

F_TEST shall detect if a lock by another process is present on the specified section.

F_LOCK and F_TLOCK shall both lock a section of a file if the section is available.

F_ULOCK shall remove locks from a section of the file.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest possible file offset shall be locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file to be locked because locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections shall be combined into a single locked section. If the request would cause the number of locks to exceed a system-imposed limit, the request shall fail.

F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available. F_LOCK shall block the calling thread until the section is available. F_TLOCK shall cause the function to fail if the section is already locked by another process.

File locks shall be released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections shall be unlocked starting at the current file offset through *size* bytes or to the end-of-file if *size* is (**off_t**)0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section shall remain locked by the process. Releasing the center portion of a locked section shall cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request shall fail.

A potential for deadlock occurs if the threads of a process controlling a locked section are blocked by accessing another process' locked section. If the system detects that deadlock would occur, *lockf*( ) shall fail with an [EDEADLK] error.

The interaction between *fcntl*( ) and *lockf*( ) locks is unspecified.

Blocking on a section shall be interrupted by any signal.

An F_ULOCK request in which *size* is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type **off_t**, when the process has an existing lock in which *size* is 0 and which includes the last byte of the requested section, shall be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise, an F_ULOCK request shall attempt to unlock only the requested section.

Attempting to lock a section of a file that is associated with a buffered stream produces unspecified results.

RETURN VALUE

Upon successful completion, *lockf*( ) shall return 0. Otherwise, it shall return −1, set *errno* to indicate an error, and existing locks shall not be changed.

ERRORS

The *lockf*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor; or *function* is F_LOCK or F_TLOCK and *fildes* is not a valid file descriptor open for writing.
[EACCES] or [EAGAIN]	The *function* argument is F_TLOCK or F_TEST and the section is already locked by another process.
[EDEADLK]	The *function* argument is F_LOCK and a deadlock is detected.
[EINTR]	A signal was caught during execution of the function.
[EINVAL]	The *function* argument is not one of F_LOCK, F_TLOCK, F_TEST, or F_ULOCK; or *size* plus the current file offset is less than 0.
[EOVERFLOW]	The offset of the first, or if *size* is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type **off_t**.

The *lockf*( ) function may fail if:

[EAGAIN]	The *function* argument is F_LOCK or F_TLOCK and the file is mapped with *mmap*( ).
[EDEADLK] or [ENOLCK]	The *function* argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request would cause the number of locks to exceed a system-imposed limit.
[EOPNOTSUPP] or [EINVAL]	The implementation does not support the locking of files of the type indicated by the *fildes* argument.

EXAMPLES

Locking a Portion of a File

In the following example, a file named **/home/cnd/mod1** is being modified. Other processes that use locking are prevented from changing it during this process. Only the first 10,000 bytes are locked, and the lock call fails if another process has any part of this area locked already.

```
#include <fcntl.h>
#include <unistd.h>

int fildes;
int status;
...
fildes = open("/home/cnd/mod1", O_RDWR);
status = lockf(fildes, F_TLOCK, (off_t)10000);
```

APPLICATION USAGE

Record-locking should not be used in combination with the *fopen*( ), *fread*( ), *fwrite*( ), and other *stdio* functions. Instead, the more primitive, non-buffered functions (such as *open*( )) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The *stdio* functions are the most common source of unexpected buffering.

The *alarm*( ) function may be used to provide a timeout facility in applications requiring it.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *chmod*( ), *close*( ), *creat*( ), *fcntl*( ), *fopen*( ), *mmap*( ), *open*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added. In particular, the description of [EINVAL] is clarified and moved from optional to mandatory status.

A note is added to the DESCRIPTION indicating the effects of attempting to lock a section of a file that is associated with a buffered stream.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

log, logf, logl — natural logarithm function

SYNOPSIS

```
#include <math.h>

double log(double x);
float logf(float x);
long double logl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the natural logarithm of their argument x, $\log_e(x)$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the natural logarithm of x.

If x is ±0, a pole error shall occur and *log*( ), *logf*( ), and *logl*( ) shall return −HUGE_VAL, −HUGE_VALF, and −HUGE_VALL, respectively.

MX For finite values of x that are less than 0, or if x is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If x is NaN, a NaN shall be returned.

If x is 1, +0 shall be returned.

If x is +Inf, x shall be returned.

ERRORS

These functions shall fail if:

Domain Error	MX The finite value of x is negative, or x is −Inf. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
Pole Error	The value of x is zero. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *log10*( ), *log1p*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *logf*( ) and *logl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

log10, log10f, log10l — base 10 logarithm function

SYNOPSIS

```
#include <math.h>

double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the base 10 logarithm of their argument x, $\log_{10}(x)$. |

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the base 10 logarithm of x.

If x is ±0, a pole error shall occur and *log10*( ), *log10f*( ), and *log10l*( ) shall return −HUGE_VAL, −HUGE_VALF, and −HUGE_VALL, respectively.

MX For finite values of x that are less than 0, or if x is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If x is NaN, a NaN shall be returned.

If x is 1, +0 shall be returned.

If x is +Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error The finite value of x is negative, or x is −Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised. |

Pole Error The value of x is zero.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *log*( ), *pow*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *log10f*( ) and *log10l*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

log1p, log1pf, log1pl — compute a natural logarithm

SYNOPSIS

```
#include <math.h>

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute $\log_e(1.0 + x)$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the natural logarithm of 1.0 + *x*.

If *x* is –1, a pole error shall occur and *log1p*( ), *log1pf*( ), and *log1pl*( ) shall return –HUGE_VAL, –HUGE_VALF, and –HUGE_VALL, respectively.

MX For finite values of *x* that are less than –1, or if *x* is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or +Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The finite value of *x* is less than –1, or *x* is –Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Pole Error — The value of *x* is –1.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

These functions may fail if:

MX Range Error — The value of *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The *log1p*( ) function is no longer marked as an extension.

The *log1pf*( ) and *log1pl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

log2()

NAME

log2, log2f, log2l — compute base 2 logarithm functions

SYNOPSIS

```
#include <math.h>

double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the base 2 logarithm of their argument x, $\log_2(x)$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the base 2 logarithm of x.

If x is ±0, a pole error shall occur and *log2*( ), *log2f*( ), and *log2l*( ) shall return –HUGE_VAL, –HUGE_VALF, and –HUGE_VALL, respectively.

MX For finite values of x that are less than 0, or if x is –Inf a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If x is NaN, a NaN shall be returned.

If x is 1, +0 shall be returned.

If x is +Inf, x shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error	The finite value of x is less than zero, or x is –Inf. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
Pole Error	The value of x is zero. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *log*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

logb, logbf, logbl — radix-independent exponent

SYNOPSIS

```
#include <math.h>

double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the exponent of *x*, which is the integral part of $\log_r |x|$, as a signed floating-point value, for non-zero *x*, where *r* is the radix of the machine's floating-point arithmetic, which is the value of FLT_RADIX defined in the **<float.h>** header.

If *x* is subnormal it is treated a though it were normalized; thus for finite positive *x*:

$$1 <= x * \texttt{FLT\_RADIX}^{-\texttt{logb}(x)} < \texttt{FLT\_RADIX}$$

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the exponent of *x*.

If *x* is ±0, a pole error shall occur and *logb*( ), *logbf*( ), and *logbl*( ) shall return −HUGE_VAL, −HUGE_VALF, and −HUGE_VALL, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±Inf, +Inf shall be returned.

ERRORS

These functions shall fail if:

Pole Error The value of *x* is ±0.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *ilogb*( ), *scalb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *logb*( ) function is no longer marked as an extension.

The *logbf*( ) and *logbl*( ) functions are added for alignment with the ISO/IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899: 1999 standard.

IEC 60559: 1989 standard floating-point extensions over the ISO/IEC 9899: 1999 standard are marked.

NAME

logf — natural logarithm function

SYNOPSIS

```
#include <math.h>

float logf(float x);
```

DESCRIPTION

Refer to *log*( ).

NAME

logl — natural logarithm function

SYNOPSIS

```
#include <math.h>

long double logl(long double x);
```

DESCRIPTION

Refer to *log*( ).

NAME

longjmp — non-local goto

SYNOPSIS

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *longjmp*( ) function shall restore the environment saved by the most recent invocation of *setjmp*( ) in the same thread, with the corresponding **jmp_buf** argument. If there is no such invocation, or if the function containing the invocation of *setjmp*( ) has terminated execution in the interim, or if the invocation of *setjmp*( ) was within the scope of an identifier with variably modified type and execution has left that scope in the interim, the behavior is undefined. CX It is unspecified whether *longjmp*( ) restores the signal mask, leaves the signal mask unchanged, or restores it to its value at the time *setjmp*( ) was called.

All accessible objects have values, and all other components of the abstract machine have state (for example, floating-point status flags and open files), as of the time *longjmp*( ) was called, except that the values of objects of automatic storage duration are unspecified if they meet all the following conditions:

- They are local to the function containing the corresponding *setjmp*( ) invocation.
- They do not have volatile-qualified type.
- They are changed between the *setjmp*( ) invocation and *longjmp*( ) call.

CX As it bypasses the usual function call and return mechanisms, *longjmp*( ) shall execute correctly in contexts of interrupts, signals, and any of their associated functions. However, if *longjmp*( ) is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

The effect of a call to *longjmp*( ) where initialization of the **jmp_buf** structure was not performed in the calling thread is undefined.

RETURN VALUE

After *longjmp*( ) is completed, program execution continues as if the corresponding invocation of *setjmp*( ) had just returned the value specified by *val*. The *longjmp*( ) function shall not cause *setjmp*( ) to return 0; if *val* is 0, *setjmp*( ) shall return 1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications whose behavior depends on the value of the signal mask should not use *longjmp*( ) and *setjmp*( ), since their effect on the signal mask is unspecified, but should instead use the *siglongjmp*( ) and *sigsetjmp*( ) functions (which can save and restore the signal mask under application control).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setjmp( ), *sigaction*( ), *siglongjmp*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now explicitly makes *longjmp*( )'s effect on the signal mask unspecified.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

lrand48 — generate uniformly distributed pseudo-random non-negative long integers

SYNOPSIS

XSI

```
#include <stdlib.h>

long lrand48(void);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

lrint, lrintf, lrintl — round to nearest integer value using current rounding direction

SYNOPSIS

```
#include <math.h>

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long**, a domain error shall occur and an unspecified value is returned. |

If the correct value is negative and too large to represent as a **long**, a domain error shall occur and an unspecified value is returned. |

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions provide floating-to-integer conversions. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the invalid floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the inexact

floating-point exception.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *llrint*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

lround, lroundf, lroundl — round to nearest integer value

SYNOPSIS

```
#include <math.h>

long lround(double x);
long lroundf(float x);
long lroundl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a domain error shall occur, and an unspecified value is returned.

If *x* is +Inf, a domain error shall occur and an unspecified value is returned.

If *x* is –Inf, a domain error shall occur and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long**, a domain error shall occur and an unspecified value is returned. |

If the correct value is negative and too large to represent as a **long**, a domain error shall occur and an unspecified value is returned. |

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is NaN or ±Inf, or the correct value is not representable as an integer.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions provide floating-to-integer conversions. They round according to the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and the invalid floating-point exception is raised. When they raise no other floating-point exception and the result differs from the argument, they raise the inexact

floating-point exception.

These functions differ from the *lrint*( ) functions in the default rounding direction, with the *lround*( ) functions rounding halfway cases away from zero and needing not to raise the inexact floating-point exception for non-integer arguments that round to within the range of the return type.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *llround*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

lsearch, lfind — linear search and update

SYNOPSIS

XSI
```
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp, size_t width,
    int (*compar)(const void *, const void *));
void *lfind(const void *key, const void *base, size_t *nelp,
    size_t width, int (*compar)(const void *, const void *));

```

DESCRIPTION

The *lsearch*( ) function shall linearly search the table and return a pointer into the table for the matching entry. If the entry does not occur, it shall be added at the end of the table. The *key* argument points to the entry to be sought in the table. The *base* argument points to the first element in the table. The *width* argument is the size of an element in bytes. The *nelp* argument points to an integer containing the current number of elements in the table. The integer to which *nelp* points shall be incremented if the entry is added to the table. The *compar* argument points to a comparison function which the application shall supply (for example, *strcmp*( )). It is called with two arguments that point to the elements being compared. The application shall ensure that the function returns 0 if the elements are equal, and non-zero otherwise. |

The *lfind*( ) function shall be equivalent to *lsearch*( ), except that if the entry is not found, it is not added to the table. Instead, a null pointer is returned. |

RETURN VALUE

If the searched for entry is found, both *lsearch*( ) and *lfind*( ) shall return a pointer to it. Otherwise, *lfind*( ) shall return a null pointer and *lsearch*( ) shall return a pointer to the newly added element.

Both functions shall return a null pointer in case of error.

ERRORS

No errors are defined.

EXAMPLES

Storing Strings in a Table

This fragment reads in less than or equal to TABSIZE strings of length less than or equal to ELSIZE and stores them in a table, eliminating duplicates.

```
#include <stdio.h>
#include <string.h>
#include <search.h>

#define TABSIZE 50
#define ELSIZE 120

...
    char line[ELSIZE], tab[TABSIZE][ELSIZE];
    size_t nel = 0;
    ...
    while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
        (void) lsearch(line, tab, &nel,
            ELSIZE, (int (*)(const void *, const void *)) strcmp);
    ...
```

Finding a Matching Entry

The following example finds any line that reads `"This is a test."`.

```
#include <search.h>
#include <string.h>
...
char line[ELSIZE], tab[TABSIZE][ELSIZE];
size_t nel = 0;
char *findline;
void *entry;

findline = "This is a test.\n";

entry = lfind(findline, tab, &nel, ELSIZE, (
    int (*)(const void *, const void *)) strcmp);
```

APPLICATION USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Undefined results can occur if there is not enough room in the table to add a new item.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *tsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

lseek — move the read/write file offset

SYNOPSIS

```
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int whence);
```

DESCRIPTION

The *lseek*( ) function shall set the file offset for the open file description associated with the file descriptor *fildes*, as follows:

- If *whence* is SEEK_SET, the file offset shall be set to *offset* bytes.
- If *whence* is SEEK_CUR, the file offset shall be set to its current location plus *offset*.
- If *whence* is SEEK_END, the file offset shall be set to the size of the file plus *offset*.

The symbolic constants SEEK_SET, SEEK_CUR, and SEEK_END are defined in **<unistd.h>**.

The behavior of *lseek*( ) on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The *lseek*( ) function shall allow the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap shall return bytes with the value 0 until data is actually written into the gap.

The *lseek*( ) function shall not, by itself, extend the size of a file.

SHM If *fildes* refers to a shared memory object, the result of the *lseek*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *lseek*( ) function is unspecified.

RETURN VALUE

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, (**off_t**)−1 shall be returned, *errno* shall be set to indicate the error, and the file offset shall remain unchanged.

ERRORS

The *lseek*( ) function shall fail if:

[EBADF]	The *fildes* argument is not an open file descriptor.
[EINVAL]	The *whence* argument is not a proper value, or the resulting file offset would be negative for a regular file, block special file, or directory.
[EOVERFLOW]	The resulting file offset would be a value which cannot be represented correctly in an object of type **off_t**.
[ESPIPE]	The *fildes* argument is associated with a pipe, FIFO, or socket.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The ISO C standard includes the functions *fgetpos*( ) and *fsetpos*( ), which work on very large files by use of a special positioning type.

Although *lseek*( ) may position the file offset beyond the end of the file, this function does not itself extend the size of the file. While the only function in IEEE Std 1003.1-200x that may directly extend the size of the file is *write*( ), *truncate*( ), and *ftruncate*( ), several functions originally derived from the ISO C standard, such as *fwrite*( ), *fprintf*( ), and so on, may do so (by causing calls on *write*( )).

An invalid file offset that would cause [EINVAL] to be returned may be both implementation-defined and device-dependent (for example, memory may have few invalid values). A negative file offset may be valid for some devices in some implementations.

The POSIX.1-1990 standard did not specifically prohibit *lseek*( ) from returning a negative offset. Therefore, an application was required to clear *errno* prior to the call and check *errno* upon return to determine whether a return value of (**off_t**)−1 is a negative offset or an indication of an error condition. The standard developers did not wish to require this action on the part of a conforming application, and chose to require that *errno* be set to [EINVAL] when the resulting file offset would be negative for a regular file, block special file, or directory.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EOVERFLOW] error condition is added. This change is to support large files.

An additional [ESPIPE] error condition is added for sockets.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *lseek*( ) results are unspecified for typed memory objects.

NAME

lstat — get symbolic link status

SYNOPSIS

```
#include <sys/stat.h>

int lstat(const char *restrict path, struct stat *restrict buf);
```

DESCRIPTION

The *lstat*( ) function shall be equivalent to *stat*( ), except when *path* refers to a symbolic link. In that case *lstat*( ) shall return information about the link, while *stat*( ) shall return information about the file the link references.

For symbolic links, the *st_mode* member shall contain meaningful information when used with the file type macros, and the *st_size* member shall contain the length of the pathname contained in the symbolic link. File mode bits and the contents of the remaining members of the **stat** structure are unspecified. The value returned in the *st_size* member is the length of the contents of the symbolic link, and does not count any trailing null.

RETURN VALUE

Upon successful completion, *lstat*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *lstat*( ) function shall fail if:

[EACCES]	A component of the path prefix denies search permission.
[EIO]	An error occurred while reading from the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[EOVERFLOW]	The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The *lstat*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
[EOVERFLOW]	One of the members is too large to store into the structure pointed to by the *buf* argument.

EXAMPLES

Obtaining Symbolic Link Status Information

The following example shows how to obtain status information for a symbolic link named **/modules/pass1**. The structure variable *buffer* is defined for the **stat** structure. If the *path* argument specified the filename for the file pointed to by the symbolic link (**/home/cnd/mod1**), the results of calling the function would be the same as those returned by a call to the *stat*( ) function.

```
#include <sys/stat.h>

struct stat buffer;
int status;
...
status = lstat("/modules/pass1", &buffer);
```

APPLICATION USAGE

None.

RATIONALE

The *lstat*( ) function is not required to update the time-related fields if the named file is not a symbolic link. While the *st_uid*, *st_gid*, *st_atime*, *st_mtime*, and *st_ctime* members of the **stat** structure may apply to a symbolic link, they are not required to do so. No functions in IEEE Std 1003.1-200x are required to maintain any of these time fields.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *readlink*( ), *stat*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- This function is now mandatory.
- The [ELOOP] optional error condition is added.

The **restrict** keyword is added to the *lstat*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

makecontext, swapcontext — manipulate user contexts

SYNOPSIS

XSI
```
#include <ucontext.h>

void makecontext(ucontext_t *ucp, void (*func)(void),
    int argc, ...);
int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);
```

DESCRIPTION

The *makecontext*( ) function shall modify the context specified by *ucp*, which has been initialized using *getcontext*( ). When this context is resumed using *swapcontext*( ) or *setcontext*( ), program execution shall continue by calling *func*, passing it the arguments that follow *argc* in the *makecontext*( ) call.

Before a call is made to *makecontext*( ), the application shall ensure that the context being modified has a stack allocated for it. The application shall ensure that the value of *argc* matches the number of integer arguments passed to *func*; otherwise, the behavior is undefined.

The *uc_link* member is used to determine the context that shall be resumed when the context being modified by *makecontext*( ) returns. The application shall ensure that the *uc_link* member is initialized prior to the call to *makecontext*( ).

The *swapcontext*( ) function shall save the current context in the context structure pointed to by *oucp* and shall set the context to the context structure pointed to by *ucp*.

RETURN VALUE

Upon successful completion, *swapcontext*( ) shall return 0. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *swapcontext*( ) function shall fail if:

[ENOMEM] The *ucp* argument does not have enough stack left to complete the operation.

EXAMPLES

The following example illustrates the use of *makecontext*( ):

```
#include <stdio.h>
#include <ucontext.h>

static ucontext_t ctx[3];

static void
f1 (void)
{
    puts("start f1");
    swapcontext(&ctx[1], &ctx[2]);
    puts("finish f1");
}

static void
f2 (void)
{
    puts("start f2");
    swapcontext(&ctx[2], &ctx[1]);
```

```
    puts("finish f2");
}

int
main (void)
{
    char st1[8192];
    char st2[8192];

    getcontext(&ctx[1]);
    ctx[1].uc_stack.ss_sp = st1;
    ctx[1].uc_stack.ss_size = sizeof st1;
    ctx[1].uc_link = &ctx[0];
    makecontext(&ctx[1], f1, 0);

    getcontext(&ctx[2]);
    ctx[2].uc_stack.ss_sp = st2;
    ctx[2].uc_stack.ss_size = sizeof st2;
    ctx[2].uc_link = &ctx[1];
    makecontext(&ctx[2], f2, 0);

    swapcontext(&ctx[0], &ctx[2]);
    return 0;
}
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *getcontext*( ), *sigaction*( ), *sigprocmask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ucontext.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the ERRORS section, the description of [ENOMEM] is changed to apply to *swapcontext*( ) only.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *swapcontext*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

malloc — a memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *malloc*( ) function shall allocate unused space for an object whose size in bytes is specified by *size* and whose value is unspecified.

The order and contiguity of storage allocated by successive calls to *malloc*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: the value returned shall be either a null pointer or a unique pointer.

RETURN VALUE

Upon successful completion with *size* not equal to 0, *malloc*( ) shall return a pointer to the allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully passed to *free*( ) shall be returned. Otherwise, it shall return a null pointer CX and set *errno* to indicate the error.

ERRORS

The *malloc*( ) function shall fail if:

CX [ENOMEM] Insufficient storage space is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *free*( ), *realloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set *errno* to indicate an error is added.
- The [ENOMEM] error condition is added.

NAME

mblen — get number of bytes in a character

SYNOPSIS

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is not a null pointer, *mblen*( ) shall determine the number of bytes constituting the character pointed to by *s*. Except that the shift state of *mbtowc*( ) is not affected, it shall be equivalent to:

```
mbtowc((wchar_t *)0, s, n);
```

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *mblen*( ).

The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function shall be placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer shall cause this function to return a non-zero value if encodings have state dependency, and 0 otherwise. If the implementation employs special bytes to change the shift state, these bytes shall not produce separate wide-character codes, but shall be grouped with an adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be unspecified.

RETURN VALUE

If *s* is a null pointer, *mblen*( ) shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mblen*( ) shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the character (if the next *n* or fewer bytes form a valid character), or return −1 (if they do not form a valid character) CX and may set *errno* to indicate the error. In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

ERRORS

The *mblen*( ) function may fail if:

XSI [EILSEQ] Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbtowc( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

NAME

mbrlen — get number of bytes in a character (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbrlen(const char *restrict s, size_t n,
    mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is not a null pointer, *mbrlen*( ) shall determine the number of bytes constituting the character pointed to by *s*. It shall be equivalent to:

```
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If *ps* is a null pointer, the *mbrlen*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *mbrlen*( ).

The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *mbrlen*( ) function shall return the first of the following that applies:

0	If the next *n* or fewer bytes complete the character that corresponds to the null wide character.
positive	If the next *n* or fewer bytes complete a valid character; the value returned shall be the number of bytes that complete the character.
(**size_t**)–2	If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed. When *n* has at least the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
(**size_t**)–1	If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character. In this case, [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

ERRORS

The *mbrlen*( ) function may fail if:

[EINVAL]	*ps* points to an object that contains an invalid conversion state.
[EILSEQ]	Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbrlen*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

mbrtowc — convert a character to a wide-character code (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s,
    size_t n, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is a null pointer, the *mbrtowc*( ) function shall be equivalent to the call:

```
mbrtowc(NULL, "", 1, ps)
```

In this case, the values of the arguments *pwc* and *n* are ignored.

If *s* is not a null pointer, the *mbrtowc*( ) function shall inspect at most *n* bytes beginning at the byte pointed to by *s* to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it shall determine the value of the corresponding wide character and then, if *pwc* is not a null pointer, shall store that value in the object pointed to by *pwc*. If the corresponding wide character is the null wide character, the resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *mbrtowc*( ) function shall use its own internal **mbstate_t** object, which shall be initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *mbrtowc*( ).

The behavior of this function is affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *mbrtowc*( ) function shall return the first of the following that applies:

0	If the next *n* or fewer bytes complete the character that corresponds to the null wide character (which is the value stored).
between 1 and *n* inclusive	If the next *n* or fewer bytes complete a valid character (which is the value stored); the value returned shall be the number of bytes that complete the character.
(**size_t**)–2	If the next *n* bytes contribute to an incomplete but potentially valid character, and all *n* bytes have been processed (no value is stored). When *n* has at least the value of the {MB_CUR_MAX} macro, this case can only occur if *s* points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
(**size_t**)–1	If an encoding error occurs, in which case the next *n* or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, [EILSEQ] shall be stored in *errno* and the conversion state is undefined.

ERRORS

The *mbrtowc*( ) function may fail if:

CX	[EINVAL]	*ps* points to an object that contains an invalid conversion state.
	[EILSEQ]	Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbrtowc*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

NAME

mbsinit — determine conversion object status

SYNOPSIS

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *ps* is not a null pointer, the *mbsinit*( ) function shall determine whether the object pointed to by *ps* describes an initial conversion state.

RETURN VALUE

The *mbsinit*( ) function shall return non-zero if *ps* is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it shall return zero.

If an **mbstate_t** object is altered by any of the functions described as ‘‘restartable’’, and is then used with a different character sequence, or in the other conversion direction, or with a different *LC_CTYPE* category setting than on earlier function calls, the behavior is undefined.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or *vice versa*) under the rules of a particular setting of the *LC_CTYPE* category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate conversion involving any character sequence, in any *LC_CTYPE* category setting.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbrlen( ), *mbrtowc*( ), *wcrtomb*( ), *mbsrtowcs*( ), *wcsrtombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

NAME

mbsrtowcs — convert a character string to a wide-character string (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
    size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *mbsrtowcs*( ) function shall convert a sequence of characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. If *dst* is not a null pointer, the converted characters shall be stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which shall also be stored. Conversion shall stop early in either of the following cases:

- A sequence of bytes is encountered that does not form a valid character.
- *len* codes have been stored into the array pointed to by *dst* (and *dst* is not a null pointer).

Each conversion shall take place as if by a call to the *mbrtowc*( ) function.

If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if *dst* is not a null pointer, the resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *mbsrtowcs*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation behaves as if no function defined in this volume of IEEE Std 1003.1-200x calls *mbsrtowcs*( ).

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the *mbsrtowcs*( ) function stores the value of the macro [EILSEQ] in *errno* and shall return (**size_t**)−1; the conversion state is undefined. Otherwise, it shall return the number of characters successfully converted, not including the terminating null (if any).

ERRORS

The *mbsrtowcs*( ) function may fail if:

CX [EINVAL]	*ps* points to an object that contains an invalid conversion state.
[EILSEQ]	Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *mbrtowc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *mbsrtowcs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard. |

The [EINVAL] error condition is marked CX. |

NAME

mbstowcs — convert a character string to a wide-character string

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *mbstowcs*( ) function shall convert a sequence of characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding wide-character codes and shall store not more than *n* wide-character codes into the array pointed to by *pwcs*. No characters that follow a null byte (which is converted into a wide-character code with value 0) shall be examined or converted. Each character shall be converted as if by a call to *mbtowc*( ), except that the shift state of *mbtowc*( ) is not affected.

No more than *n* elements shall be modified in the array pointed to by *pwcs*. If copying takes place between objects that overlap, the behavior is undefined.

XSI The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale. If *pwcs* is a null pointer, *mbstowcs*( ) shall return the length required to convert the entire array regardless of the value of *n*, but no values are stored.

RETURN VALUE

CX XSI If an invalid character is encountered, *mbstowcs*( ) shall return (**size_t**)–1 and may set *errno* to indicate the error. Otherwise, *mbstowcs*( ) shall return the number of the array elements modified (or required if *pwcs* is null), not including a terminating 0 code, if any. The array shall not be zero-terminated if the value returned is *n*.

ERRORS

The *mbstowcs*( ) function may fail if:

XSI [EILSEQ] Invalid byte sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mblen( ), *mbtowc*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The *mbstowcs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

Extensions beyond the ISO C standard are now marked.

NAME

mbtowc — convert a character to a wide-character code

SYNOPSIS

```
#include <stdlib.h>

int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is not a null pointer, *mbtowc*( ) shall determine the number of the bytes that constitute the character pointed to by *s*. It shall then determine the wide-character code for the value of type **wchar_t** that corresponds to that character. (The value of the wide-character code corresponding to the null byte is 0.) If the character is valid and *pwc* is not a null pointer, *mbtowc*( ) shall store the wide-character code in the object pointed to by *pwc*.

The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer shall cause this function to return a non-zero value if encodings have state dependency, and 0 otherwise. If the implementation employs special bytes to change the shift state, these bytes shall not produce separate wide-character codes, but shall be grouped with an adjacent character. Changing the *LC_CTYPE* category causes the shift state of this function to be unspecified. At most *n* bytes of the array pointed to by *s* shall be examined.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *mbtowc*( ).

RETURN VALUE

If *s* is a null pointer, *mbtowc*( ) shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *mbtowc*( ) shall either return 0 (if *s* points to the null byte), or return the number of bytes that constitute the converted character (if the next *n* or fewer bytes form a valid character), or return −1 CX and may set *errno* to indicate the error (if they do not form a valid character).

In no case shall the value returned be greater than *n* or the value of the {MB_CUR_MAX} macro.

ERRORS

The *mbtowc*( ) function may fail if:

XSI [EILSEQ] Invalid character sequence is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mblen( ), *mbstowcs*( ), *wctomb*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Aligned with the ISO C standard.

Issue 6

The *mbtowc*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

Extensions beyond the ISO C standard are now marked.

NAME

memccpy — copy bytes in memory

SYNOPSIS

XSI

```
#include <string.h>

void *memccpy(void *restrict s1, const void *restrict s2,
    int c, size_t n);
```

DESCRIPTION

The *memccpy*( ) function shall copy bytes from memory area *s2* into *s1*, stopping after the first occurrence of byte *c* (converted to an **unsigned char**) is copied, or after *n* bytes are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *memccpy*( ) function shall return a pointer to the byte after the copy of *c* in *s1*, or a null pointer if *c* was not found in the first *n* bytes of *s2*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *memccpy*( ) function does not check for the overflow of the receiving memory area.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The **restrict** keyword is added to the *memccpy*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

memchr — find byte in memory

SYNOPSIS

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *memchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in the initial *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s*.

RETURN VALUE

The *memchr*( ) function shall return a pointer to the located byte, or a null pointer if the byte does not occur in the object.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

memcmp — compare bytes in memory

SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *memcmp*( ) function shall compare the first *n* bytes (each interpreted as **unsigned char**) of the object pointed to by *s1* to the first *n* bytes of the object pointed to by *s2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the objects being compared.

RETURN VALUE

The *memcmp*( ) function shall return an integer greater than, equal to, or less than 0, if the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

memcpy — copy bytes in memory

SYNOPSIS

```
#include <string.h>

void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *memcpy*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *memcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *memcpy*( ) function does not check for the overflowing of the receiving memory area.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *memcpy*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

memmove — copy bytes in memory with overlapping areas

SYNOPSIS

```
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *memmove*( ) function shall copy *n* bytes from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* bytes from the object pointed to by *s2* are first copied into a temporary array of *n* bytes that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* bytes from the temporary array are copied into the object pointed to by *s1*.

RETURN VALUE

The *memmove*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

NAME

memset — set bytes in memory

SYNOPSIS

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *memset*( ) function shall copy *c* (converted to an **unsigned char**) into each of the first *n* bytes of the object pointed to by *s*.

RETURN VALUE

The *memset*( ) function shall return *s*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

mkdir — make a directory

SYNOPSIS

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

DESCRIPTION

The *mkdir*( ) function shall create a new directory with name *path*. The file permission bits of the new directory shall be initialized from *mode*. These file permission bits of the *mode* argument shall be modified by the process' file creation mask.

When bits in *mode* other than the file permission bits are set, the meaning of these additional bits is implementation-defined.

The directory's user ID shall be set to the process' effective user ID. The directory's group ID shall be set to the group ID of the parent directory or to the effective group ID of the process. Implementations shall provide a way to initialize the directory's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the directory's group ID to the effective group ID of the calling process.

The newly created directory shall be an empty directory.

If *path* names a symbolic link, *mkdir*( ) shall fail and set *errno* to [EEXIST].

Upon successful completion, *mkdir*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the directory. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

RETURN VALUE

Upon successful completion, *mkdir*( ) shall return 0. Otherwise, −1 shall be returned, no directory shall be created, and *errno* shall be set to indicate the error.

ERRORS

The *mkdir*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be created.
[EEXIST]	The named file exists.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[EMLINK]	The link count of the parent directory would exceed {LINK_MAX}.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.
[ENOSPC]	The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The parent directory resides on a read-only file system.

The *mkdir*( ) function may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Creating a Directory

The following example shows how to create a directory named **/home/cnd/mod1**, with read/write/search permissions for owner and group, and with read/search permissions for others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

APPLICATION USAGE

None.

RATIONALE

The *mkdir*( ) function originated in 4.2 BSD and was added to System V in Release 3.0.

4.3 BSD detects [ENAMETOOLONG].

The POSIX.1-1990 standard required that the group ID of a newly created directory be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the directory is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

umask( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

mkfifo — make a FIFO special file

SYNOPSIS

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

DESCRIPTION

The *mkfifo*( ) function shall create a new FIFO special file named by the pathname pointed to by *path*. The file permission bits of the new FIFO shall be initialized from *mode*. The file permission bits of the *mode* argument shall be modified by the process' file creation mask.

When bits in *mode* other than the file permission bits are set, the effect is implementation-defined.

If *path* names a symbolic link, *mkfifo*( ) shall fail and set *errno* to [EEXIST].

The FIFO's user ID shall be set to the process' effective user ID. The FIFO's group ID shall be set to the group ID of the parent directory or to the effective group ID of the process. Implementation shall provide a way to initialize the FIFO's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the FIFO's group ID to the effective group ID of the calling process.

Upon successful completion, *mkfifo*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned, no FIFO shall be created, and *errno* shall be set to indicate the error.

ERRORS

The *mkfifo*( ) function shall fail if:

[EACCES]	A component of the path prefix denies search permission, or write permission is denied on the parent directory of the FIFO to be created.
[EEXIST]	The named file already exists.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.
[ENOSPC]	The directory that would contain the new file cannot be extended or the file system is out of file-allocation resources.
[ENOTDIR]	A component of the path prefix is not a directory.
[EROFS]	The named file resides on a read-only file system.

The *mkfifo*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}. |

EXAMPLES

Creating a FIFO File

The following example shows how to create a FIFO file named **/home/cnd/mod_done**, with read/write permissions for owner, and with read permissions for group and others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
    S_IRGRP | S_IROTH);
```

APPLICATION USAGE

None.

RATIONALE

The syntax of this function is intended to maintain compatibility with historical implementations of *mknod*( ). The latter function was included in the 1984 /usr/group standard but only for use in creating FIFO special files. The *mknod*( ) function was originally excluded from the POSIX.1-1988 standard as implementation-defined and replaced by *mkdir*( ) and *mkfifo*( ). The *mknod*( ) function is now included for alignment with the Single UNIX Specification.

The POSIX.1-1990 standard required that the group ID of a newly created FIFO be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the FIFO is created, or determine under what conditions the implementation will set the desired group ID. |

FUTURE DIRECTIONS

None.

SEE ALSO

umask( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

mknod — make a directory, a special or regular file

SYNOPSIS

XSI
```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev);
```

DESCRIPTION

The *mknod*( ) function shall create a new file named by the pathname to which the argument *path* points. |

The file type for *path* is OR'ed into the *mode* argument, and the application shall select one of the following symbolic constants:

Name	Description
S_IFIFO	FIFO-special
S_IFCHR	Character-special (non-portable)
S_IFDIR	Directory (non-portable)
S_IFBLK	Block-special (non-portable)
S_IFREG	Regular (non-portable)

The only portable use of *mknod*( ) is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is not 0, the behavior of *mknod*( ) is unspecified.

The permissions for the new file are OR'ed into the *mode* argument, and may be selected from any combination of the following symbolic constants:

Name	Description
S_ISUID	Set user ID on execution.
S_ISGID	Set group ID on execution.
S_IRWXU	Read, write, or execute (search) by owner.
S_IRUSR	Read by owner.
S_IWUSR	Write by owner.
S_IXUSR	Execute (search) by owner.
S_IRWXG	Read, write, or execute (search) by group.
S_IRGRP	Read by group.
S_IWGRP	Write by group.
S_IXGRP	Execute (search) by group.
S_IRWXO	Read, write, or execute (search) by others.
S_IROTH	Read by others.
S_IWOTH	Write by others.
S_IXOTH	Execute (search) by others.
S_ISVTX	On directories, restricted deletion flag.

The user ID of the file shall be initialized to the effective user ID of the process. The group ID of the file shall be initialized to either the effective group ID of the process or the group ID of the parent directory. Implementations shall provide a way to initialize the file's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the file's gorup ID to the effective group ID of the calling proces. |

The owner, group, and other permission bits of *mode* shall be modified by the file mode creation mask of the process. The *mknod*( ) function shall clear each bit whose corresponding bit in the file mode creation mask of the process is set.

If *path* names a symbolic link, *mknod*( ) shall fail and set *errno* to [EEXIST].

Upon successful completion, *mknod*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file. Also, the *st_ctime* and *st_mtime* fields of the directory that contains the new entry shall be marked for update.

Only a process with appropriate privileges may invoke *mknod*( ) for file types other than FIFO-special.

RETURN VALUE

Upon successful completion, *mknod*( ) shall return 0. Otherwise, it shall return −1, the new file shall not be created, and *errno* shall be set to indicate the error.

ERRORS

The *mknod*( ) function shall fail if:

[EACCES]	A component of the path prefix denies search permission, or write permission is denied on the parent directory.
[EEXIST]	The named file exists.
[EINVAL]	An invalid argument exists.
[EIO]	An I/O error occurred while accessing the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of a pathname exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of the path prefix specified by *path* does not name an existing directory or *path* is an empty string.
[ENOSPC]	The directory that would contain the new file cannot be extended or the file system is out of file allocation resources.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The invoking process does not have appropriate privileges and the file type is not FIFO-special.
[EROFS]	The directory in which the file is to be created is located on a read-only file system.

The *mknod*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

Creating a FIFO Special File

The following example shows how to create a FIFO special file named **/home/cnd/mod_done**, with read/write permissions for owner, and with read permissions for group and others.

```
#include <sys/types.h>
#include <sys/stat.h>

dev_t dev;
int   status;
...
status  = mknod("/home/cnd/mod_done", S_IFIFO | S_IWUSR |
    S_IRUSR | S_IRGRP | S_IROTH, dev);
```

APPLICATION USAGE

mkfifo( ) is preferred over this function for making FIFO special files.

RATIONALE

The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the file is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *exec*, *mkdir*( ), *mkfifo*( ), *open*( ), *stat*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

mkstemp — make a unique filename

SYNOPSIS

XSI
```
#include <stdlib.h>

int mkstemp(char *template);
```

DESCRIPTION

The *mkstemp*( ) function shall replace the contents of the string pointed to by *template* by a unique filename, and return a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in *template* should look like a filename with six trailing *X*s; *mkstemp*( ) replaces each *X* with a character from the portable filename character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file at the time of a call to *mkstemp*( ).

RETURN VALUE

Upon successful completion, *mkstemp*( ) shall return an open file descriptor. Otherwise, –1 shall be returned if no suitable file could be created.

ERRORS

No errors are defined.

EXAMPLES

Generating a Filename

The following example creates a file with a 10-character name beginning with the characters `"file"` and opens the file for reading and writing. The value returned as the value of *fd* is a file descriptor that identifies the file.

```
#include <stdlib.h>
...
char template[] = "/tmp/fileXXXXXX";
int fd;

fd = mkstemp(template);
```

APPLICATION USAGE

It is possible to run out of letters.

The *mkstemp*( ) function need not check to determine whether the filename part of *template* exceeds the maximum allowable filename length.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpid( ), *open*( ), *tmpfile*( ), *tmpnam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

mktemp — make a unique filename (**LEGACY**)

SYNOPSIS

XSI

```
#include <stdlib.h>

char *mktemp(char *template);
```

DESCRIPTION

The *mktemp*( ) function shall replace the contents of the string pointed to by *template* by a unique filename and return *template*. The application shall initialize *template* to be a filename with six trailing *X*s; *mktemp*( ) shall replace each *X* with a single byte character from the portable filename character set.

RETURN VALUE

The *mktemp*( ) function shall return the pointer *template*. If a unique name cannot be created, *template* shall point to a null string.

ERRORS

No errors are defined.

EXAMPLES

Generating a Filename

The following example replaces the contents of the `"template"` string with a 10-character filename beginning with the characters `"file"` and returns a pointer to the `"template"` string that contains the new filename.

```
#include <stdlib.h>
...
char *template = "/tmp/fileXXXXXX";
char *ptr;

ptr = mktemp(template);
```

APPLICATION USAGE

Between the time a pathname is created and the file opened, it is possible for some other process | to create a file with the same name. The *mkstemp*( ) function avoids this problem and is preferred over this function.

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

mkstemp( ), *tmpfile*( ), *tmpnam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

mktime — convert broken-down time into time since the Epoch

SYNOPSIS

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *mktime*( ) function shall convert the broken-down time, expressed as local time, in the structure pointed to by *timeptr*, into a time since the Epoch value with the same encoding as that of the values returned by *time*( ). The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to the ranges described in **<time.h>**.

CX A positive or 0 value for *tm_isdst* shall cause *mktime*( ) to presume initially that Daylight Savings Time, respectively, is or is not in effect for the specified time. A negative value for *tm_isdst* shall cause *mktime*( ) to attempt to determine whether Daylight Saving Time is in effect for the specified time.

Local timezone information shall be set as though *mktime*( ) called *tzset*( ).

The relationship between the **tm** structure (defined in the **<time.h>** header) and the time in seconds since the Epoch is that the result shall be as specified in the expression given in the definition of seconds since the Epoch (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch) corrected for timezone and any seasonal time adjustments, where the names in the structure and in the expression correspond.

Upon successful completion, the values of the *tm_wday* and *tm_yday* components of the structure shall be set appropriately, and the other components are set to represent the specified time since the Epoch, but with their values forced to the ranges indicated in the **<time.h>** entry; the final value of *tm_mday* shall not be set until *tm_mon* and *tm_year* are determined.

RETURN VALUE

The *mktime*( ) function shall return the specified time since the Epoch encoded as a value of type **time_t**. If the time since the Epoch cannot be represented, the function shall return the value (**time_t**)–1.

ERRORS

No errors are defined.

EXAMPLES

What day of the week is July 4, 2001?

```
#include <stdio.h>
#include <time.h>

struct tm time_str;

char daybuf[20];

int main(void)
{
    time_str.tm_year = 2001 – 1900;
    time_str.tm_mon = 7 – 1;
    time_str.tm_mday = 4;
```

```
        time_str.tm_hour = 0;
        time_str.tm_min = 0;
        time_str.tm_sec = 1;
        time_str.tm_isdst = −1;
        if (mktime(&time_str) == -1)
            (void)puts("-unknown-");
        else {
            (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str);
            (void)puts(daybuf);
        }
        return 0;
    }
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *strftime*( ), *strptime*( ), *time*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard and the ANSI C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

mlock, munlock — lock or unlock a range of process address space (**REALTIME**)

SYNOPSIS

MLR
```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

DESCRIPTION

The *mlock*( ) function shall cause those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes to be memory-resident until unlocked or until the process exits or *exec*s another process image. The implementation may require that *addr* be a multiple of {PAGESIZE}.

The *munlock*( ) function shall unlock those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes, regardless of how many times *mlock*( ) has been called by the process for any of the pages in the specified range. The implementation may require that *addr* be a multiple of {PAGESIZE}. |

If any of the pages in the range specified to a call to *munlock*( ) are also mapped into the address spaces of other processes, any locks established on those pages by another process are unaffected by the call of this process to *munlock*( ). If any of the pages in the range specified by a call to *munlock*( ) are also mapped into other portions of the address space of the calling process outside the range specified, any locks established on those pages via the other mappings are also unaffected by this call.

Upon successful return from *mlock*( ), pages in the specified range shall be locked and memory-resident. Upon successful return from *munlock*( ), pages in the specified range shall be unlocked with respect to the address space of the process. Memory residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with *mlock*( ).

RETURN VALUE

Upon successful completion, the *mlock*( ) and *munlock*( ) functions shall return a value of zero. Otherwise, no change is made to any locks in the address space of the process, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mlock*( ) and *munlock*( ) functions shall fail if:

[ENOMEM] Some or all of the address range specified by the *addr* and *len* arguments does not correspond to valid mapped pages in the address space of the process.

The *mlock*( ) function shall fail if:

[EAGAIN] Some or all of the memory identified by the operation could not be locked when the call was made.

The *mlock*( ) and *munlock*( ) functions may fail if:

[EINVAL] The *addr* argument is not a multiple of {PAGESIZE}.

The *mlock*( ) function may fail if:

[ENOMEM] Locking the pages mapped by the specified range would exceed an implementation-defined limit on the amount of memory that the process may lock.

[EPERM] The calling process does not have the appropriate privilege to perform the requested operation.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *mlockall*( ), *munmap*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mlock*( ) and *munlock*( ) functions are marked as part of the Range Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Range Memory Locking option.

NAME

mlockall, munlockall — lock/unlock the address space of a process (**REALTIME**)

SYNOPSIS

ML
```
#include <sys/mman.h>

int mlockall(int flags);
int munlockall(void);
```

DESCRIPTION

The *mlockall*( ) function shall cause all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or *exec*s another process image. The *flags* argument determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both. The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants, defined in **<sys/mman.h>**:

MCL_CURRENT	Lock all of the pages currently mapped into the address space of the process.
MCL_FUTURE	Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

If MCL_FUTURE is specified, and the automatic locking of future mappings eventually causes the amount of locked memory to exceed the amount of available physical memory or any other implementation-defined limit, the behavior is implementation-defined. The manner in which the implementation informs the application of these situations is also implementation-defined.

The *munlockall*( ) function shall unlock all currently mapped pages of the address space of the process. Any pages that become mapped into the address space of the process after a call to *munlockall*( ) shall not be locked, unless there is an intervening call to *mlockall*( ) specifying MCL_FUTURE or a subsequent call to *mlockall*( ) specifying MCL_CURRENT. If pages mapped into the address space of the process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes shall be unaffected by a call by this process to *munlockall*( ). |

Upon successful return from the *mlockall*( ) function that specifies MCL_CURRENT, all currently mapped pages of the process' address space shall be memory-resident and locked. Upon return from the *munlockall*( ) function, all currently mapped pages of the process' address space shall be unlocked with respect to the process' address space. The memory residency of unlocked pages is unspecified.

The appropriate privilege is required to lock process memory with *mlockall*( ).

RETURN VALUE

Upon successful completion, the *mlockall*( ) function shall return a value of zero. Otherwise, no additional memory shall be locked, and the function shall return a value of −1 and set *errno* to indicate the error. The effect of failure of *mlockall*( ) on previously existing locks in the address space is unspecified.

If it is supported by the implementation, the *munlockall*( ) function shall always return a value of zero. Otherwise, the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mlockall*( ) function shall fail if:

[EAGAIN]	Some or all of the memory identified by the operation could not be locked when the call was made.

[EINVAL]	The *flags* argument is zero, or includes unimplemented flags.

The *mlockall*( ) function may fail if:

[ENOMEM]	Locking all of the pages currently mapped into the address space of the process would exceed an implementation-defined limit on the amount of memory that the process may lock.
[EPERM]	The calling process does not have the appropriate privilege to perform the requested operation.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *mlock*( ), *munmap*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mlockall*( ) and *munlockall*( ) functions are marked as part of the Process Memory Locking option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Memory Locking option.

NAME

mmap — map pages of memory

SYNOPSIS

MF|SHM

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
    int fildes, off_t off);
```

DESCRIPTION

TYM The *mmap*( ) function shall establish a mapping between a process' address space and a file, shared memory object, or typed memory object. The format of the call is as follows:

```
pa=mmap(addr, len, prot, flags, fildes, off);
```

The *mmap*( ) function shall establish a mapping between the address space of the process at an address *pa* for *len* bytes to the memory object represented by the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an implementation-defined function of the parameter *addr* and the values of *flags*, further described below. A successful *mmap*( ) call shall return *pa* as its result. The address range starting at *pa* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at *off* and continuing for *len* bytes shall be legitimate for the possible (not necessarily current) offsets in the TYM file, shared memory object, or typed memory object represented by *fildes*.

TYM If *fildes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the memory object to be mapped shall be that portion of the typed memory object allocated by the implementation as specified below. In this case, if *off* is non-zero, the behavior of *mmap*( ) is undefined. If *fildes* refers to a valid typed memory object that is not accessible from the calling process, *mmap*( ) shall fail.

The mapping established by *mmap*( ) shall replace any previous mappings for those whole pages containing any part of the address space of the process starting at *pa* and continuing for *len* bytes.

If the size of the mapped file changes after the call to *mmap*( ) as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

TYM The *mmap*( ) function shall be supported for regular files, shared memory objects, and typed memory objects. Support for any other type of file is unspecified.

The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* shall be either PROT_NONE or the bitwise-inclusive OR of one or more of the other flags in the following table, defined in the **<sys/mman.h>** header.

Symbolic Constant	Description
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data cannot be accessed.

If an implementation cannot support the combination of access types specified by *prot*, the call MPR to *mmap*( ) shall fail. An implementation may permit accesses other than those specified by *prot*; however, if the Memory Protection option is supported, the implementation shall not permit a write to succeed where PROT_WRITE has not been set or shall not permit any access where PROT_NONE alone has been set. The implementation shall support at least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and PROT_WRITE. If the Memory Protection option is not supported, the result of any access that conflicts with the specified protection is undefined. The file descriptor *fildes* shall have been opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the application shall ensure that it has opened the file descriptor *fildes* with write permission unless MAP_PRIVATE is specified in the *flags* parameter as described below.

The parameter *flags* provides other information about the handling of the mapped data. The value of *flags* is the bitwise-inclusive OR of these options, defined in **<sys/mman.h>**:

Symbolic Constant	Description
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	Interpret *addr* exactly.

Implementations that do not support the Memory Mapped Files option are not required to support MAP_PRIVATE.

XSI It is implementation-defined whether MAP_FIXED shall be supported. MAP_FIXED shall be supported on XSI-conformant systems.

MAP_SHARED and MAP_PRIVATE describe the disposition of write references to the memory object. If MAP_SHARED is specified, write references shall change the underlying object. If MAP_PRIVATE is specified, modifications to the mapped data by the calling process shall be visible only to the calling process and shall not change the underlying object. It is unspecified whether modifications to the underlying object done after the MAP_PRIVATE mapping is established are visible through the MAP_PRIVATE mapping. Either MAP_SHARED or MAP_PRIVATE can be specified, but not both. The mapping type is retained across *fork*( ). |

TYM When *fildes* represents a typed memory object opened with either the POSIX_TYPED_MEM_ALLOCATE flag or the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, *mmap*( ) shall, if there are enough resources available, map *len* bytes allocated from the corresponding typed memory object which were not previously allocated to any process in any processor that may access that typed memory object. If there are not enough resources available, the function shall fail. If *fildes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, these allocated bytes shall be contiguous within the typed memory object. If *fildes* represents a typed memory object opened with the POSIX_TYPED_MEM_ALLOCATE flag, these allocated bytes may be composed of non-contiguous fragments within the typed memory object. If *fildes* represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag nor the POSIX_TYPED_MEM_ALLOCATE flag, *len* bytes starting at offset *off* within the typed memory object are mapped, exactly as when mapping a file or shared memory object. In this case, if two processes map an area of typed memory using the same *off* and *len* values and using file descriptors that refer to the same memory pool (either from the same port or from a different port), both processes shall map the same region of storage.

When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of *pa* shall be *addr*, exactly. If MAP_FIXED is set, *mmap*( ) may return MAP_FAILED and set *errno* to [EINVAL]. If a MAP_FIXED request is successful, the mapping established by *mmap*( ) replaces any previous mappings for the process' pages in the range [*pa*,*pa*+*len*).

When MAP_FIXED is not set, the implementation uses *addr* in an implementation-defined manner to arrive at *pa*. The *pa* so chosen shall be an area of the address space that the implementation deems suitable for a mapping of *len* bytes to the file. All implementations interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping.

The *off* argument is constrained to be aligned and sized according to the value returned by *sysconf*( ) when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the application shall ensure that the argument *addr* also meets these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation shall include, in any mapping operation, any partial page specified by the range [*pa*,*pa*+*len*).

MPR The system shall always zero-fill any partial page at the end of an object. Further, the system shall never write out any modified portions of the last page of an object which are beyond its end. References within the address range starting at *pa* and continuing for *len* bytes to whole pages following the end of an object shall result in delivery of a SIGBUS signal.

An implementation may generate SIGBUS signals when a reference would cause an error in the mapped object, such as out-of-space condition.

The *mmap*( ) function shall add an extra reference to the file associated with the file descriptor *fildes* which is not removed by a subsequent *close*( ) on that file descriptor. This reference shall be removed when there are no more mappings to the file.

The *st_atime* field of the mapped file may be marked for update at any time between the *mmap*( ) call and the corresponding *munmap*( ) call. The initial read or write reference to a mapped region shall cause the file's *st_atime* field to be marked for update if it has not already been marked for update.

The *st_ctime* and *st_mtime* fields of a file that is mapped with MAP_SHARED and PROT_WRITE shall be marked for update at some point in the interval between a write reference to the mapped region and the next call to *msync*( ) with MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call and if the underlying file is modified as a result of a write reference, then these fields shall be marked for update at some time after the write reference.

There may be implementation-defined limits on the number of memory regions that can be mapped (per process or per system).

XSI If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of *shmat*( ) is implementation-defined.

If *mmap*( ) fails for reasons other than [EBADF], [EINVAL], or [ENOTSUP], some of the mappings in the address range starting at *addr* and continuing for *len* bytes may have been unmapped.

RETURN VALUE

Upon successful completion, the *mmap*( ) function shall return the address at which the mapping was placed (*pa*); otherwise, it shall return a value of MAP_FAILED and set *errno* to indicate the error. The symbol MAP_FAILED is defined in the **<sys/mman.h>** header. No successful return from *mmap*( ) shall return the value MAP_FAILED.

ERRORS

The *mmap*( ) function shall fail if:

	[EACCES]	The *fildes* argument is not open for read, regardless of the protection specified, or *fildes* is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.
ML	[EAGAIN]	The mapping could not be locked in memory, if required by *mlockall*( ), due to a lack of resources.
	[EBADF]	The *fildes* argument is not a valid open file descriptor.
	[EINVAL]	The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of the page size as returned by *sysconf*( ), or are considered invalid by the implementation.
	[EINVAL]	The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).
	[EMFILE]	The number of mapped regions would exceed an implementation-defined limit (per process or per system).
	[ENODEV]	The *fildes* argument refers to a file whose type is not supported by *mmap*( ).
	[ENOMEM]	MAP_FIXED was specified, and the range [*addr*,*addr*+*len*) exceeds that allowed for the address space of a process; or, if MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.
ML	[ENOMEM]	The mapping could not be locked in memory, if required by *mlockall*( ), because it would require more space than the system is able to supply.
		MAP_FIXED or MAP_PRIVATE was specified in the *flags* argument and the implementation does not support this functionality.
TYM	[ENOMEM]	Not enough unallocated memory resources remain in the typed memory object designated by *fildes* to allocate *len* bytes.
	[ENOTSUP]	The implementation does not support the combination of accesses requested in the *prot* argument.
	[ENXIO]	Addresses in the range [*off*,*off*+*len*) are invalid for the object specified by *fildes*.
	[ENXIO]	MAP_FIXED was specified in *flags* and the combination of *addr*, *len*, and *off* is invalid for the object specified by *fildes*.
TYM	[ENXIO]	The *fildes* argument refers to a typed memory object that is not accessible from the calling process.
	[EOVERFLOW]	The file is a regular file and the value of *off* plus *len* exceeds the offset maximum established in the open file description associated with *fildes*.

EXAMPLES

None.

APPLICATION USAGE

Use of *mmap*( ) may reduce the amount of memory available to other memory allocation functions.

Use of MAP_FIXED may result in unspecified behavior in further use of *malloc*( ) and *shmat*( ). The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of resources.

The application must ensure correct synchronization when using *mmap*( ) in conjunction with any other file access method, such as *read*( ) and *write*( ), standard input/output, and *shmat*( ).

The *mmap*( ) function allows access to resources via address space manipulations, instead of *read*( )/*write*( ). Once a file is mapped, all a process has to do to access it is use the data at the address to which the file was mapped. So, using pseudo-code to illustrate the way in which an existing program might be changed to use *mmap*( ), the following:

```
fildes = open(...)
lseek(fildes, some_offset)
read(fildes, buf, len)
/* Use data in buf. */
```

becomes:

```
fildes = open(...)
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)
/* Use data at address. */
```

The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

RATIONALE

After considering several other alternatives, it was decided to adopt the *mmap*( ) definition found in SVR4 for mapping memory objects into process address spaces. The SVR4 definition is minimal, in that it describes only what has been built, and what appears to be necessary for a general and portable mapping facility.

Note that while *mmap*( ) was first designed for mapping files, it is actually a general-purpose mapping facility. It can be used to map any appropriate object, such as memory, files, devices, and so on, into the address space of a process.

When a mapping is established, it is possible that the implementation may need to map more than is requested into the address space of the process because of hardware requirements. An application, however, cannot count on this behavior. Implementations that do not use a paged architecture may simply allocate a common memory region and return the address of it; such implementations probably do not allocate any more than is necessary. References past the end of the requested area are unspecified.

If an application requests a mapping that would overlay existing mappings in the process, it might be desirable that an implementation detect this and inform the application. However, the default, portable (not MAP_FIXED) operation does not overlay existing mappings. On the other hand, if the program specifies a fixed address mapping (which requires some implementation knowledge to determine a suitable address, if the function is supported at all), then the program is presumed to be successfully managing its own address space and should be trusted when it asks to map over existing data structures. Furthermore, it is also desirable to make as few system calls as possible, and it might be considered onerous to require an *munmap*( ) before an *mmap*( ) to the same address range. This volume of IEEE Std 1003.1-200x specifies that the new mappings replace any existing mappings, following existing practice in this regard.

It is not expected, when the Memory Protection option is supported, that all hardware implementations are able to support all combinations of permissions at all addresses. When this option is supported, implementations are required to disallow write access to mappings without write permission and to disallow access to mappings without any access permission. Other than these restrictions, implementations may allow access types other than those requested by the application. For example, if the application requests only PROT_WRITE, the implementation may also allow read access. A call to *mmap*( ) fails if the implementation cannot support allowing all the access requested by the application. For example, some implementations cannot support a request for both write access and execute access simultaneously. All implementations supporting the Memory Protection option must support requests for no access, read access, write access, and both read and write access. Strictly conforming code must only rely on the required checks. These restrictions allow for portability across a wide range of hardware.

The MAP_FIXED address treatment is likely to fail for non-page-aligned values and for certain architecture-dependent address ranges. Conforming implementations cannot count on being able to choose address values for MAP_FIXED without utilizing non-portable, implementation-defined knowledge. Nonetheless, MAP_FIXED is provided as a standard interface conforming to existing practice for utilizing such knowledge when it is available.

Similarly, in order to allow implementations that do not support virtual addresses, support for directly specifying any mapping addresses via MAP_FIXED is not required and thus a conforming application may not count on it.

The MAP_PRIVATE function can be implemented efficiently when memory protection hardware is available. When such hardware is not available, implementations can implement such ‘‘mappings’’ by simply making a real copy of the relevant data into process private memory, though this tends to behave similarly to *read*( ).

The function has been defined to allow for many different models of using shared memory. However, all uses are not equally portable across all machine architectures. In particular, the *mmap*( ) function allows the system as well as the application to specify the address at which to map a specific region of a memory object. The most portable way to use the function is always to let the system choose the address, specifying NULL as the value for the argument *addr* and not to specify MAP_FIXED.

If it is intended that a particular region of a memory object be mapped at the same address in a group of processes (on machines where this is even possible), then MAP_FIXED can be used to pass in the desired mapping address. The system can still be used to choose the desired address if the first such mapping is made without specifying MAP_FIXED, and then the resulting mapping address can be passed to subsequent processes for them to pass in via MAP_FIXED. The availability of a specific address range cannot be guaranteed, in general.

The *mmap*( ) function can be used to map a region of memory that is larger than the current size of the object. Memory access within the mapping but beyond the current end of the underlying objects may result in SIGBUS signals being sent to the process. The reason for this is that the size of the object can be manipulated by other processes and can change at any moment. The implementation should tell the application that a memory reference is outside the object where this can be detected; otherwise, written data may be lost and read data may not reflect actual data in the object.

Note that references beyond the end of the object do not extend the object as the new end cannot be determined precisely by most virtual memory hardware. Instead, the size can be directly manipulated by *ftruncate*( ).

Process memory locking does apply to shared memory regions, and the MEMLOCK_FUTURE argument to *memlockall*( ) can be relied upon to cause new shared memory regions to be automatically locked.

Existing implementations of *mmap*( ) return the value −1 when unsuccessful. Since the casting of this value to type **void *** cannot be guaranteed by the ISO C standard to be distinct from a successful value, this volume of IEEE Std 1003.1-200x defines the symbol MAP_FAILED, which a conforming implementation does not return as the result of a successful call.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fcntl*( ), *fork*( ), *lockf*( ), *msync*( ), *munmap*( ), *mprotect*( ), *posix_typed_mem_open*( ), *shmat*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *mmap*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The [EAGAIN] and [ENOTSUP] mandatory error conditions are added.
- New cases of [ENOMEM] and [ENXIO] are added as mandatory error conditions.
- The value returned on failure is the value of the constant MAP_FAILED; this was previously defined as –1.

Large File Summit extensions are added.

Issue 6

The *mmap*( ) function is marked as part of the Memory Mapped Files option.

The Open Group Corrigendum U028/6 is applied, changing (**void** *)–1 to MAP_FAILED.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to described the use of MAP_FIXED.
- The DESCRIPTION is updated to describe the addition of an extra reference to the file associated with the file descriptor passed to *mmap*( ).
- The DESCRIPTION is updated to state that there may be implementation-defined limits on the number of memory regions that can be mapped.
- The DESCRIPTION is updated to describe constraints on the alignment and size of the *off* argument.
- The [EINVAL] and [EMFILE] error conditions are added.
- The [EOVERFLOW] error condition is added. This change is to support large files.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The DESCRIPTION is updated to describe the cases when MAP_PRIVATE and MAP_FIXED need not be supported.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- New [ENOMEM] and [ENXIO] errors are added to the ERRORS section.
- The *posix_typed_mem_open*( ) function is added to the SEE ALSO section.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

modf, modff, modfl — decompose a floating-point number

SYNOPSIS

```
#include <math.h>

double modf(double x, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall break the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a **double** (for the *modf*( ) function), a **float** (for the *modff*( ) function), or a **long double** (for the *modfl*( ) function), in the object pointed to by *iptr*.

RETURN VALUE

Upon successful completion, these functions shall return the signed fractional part of *x*.

MX If *x* is NaN, a NaN shall be returned, and **iptr* shall be set to a NaN.

If *x* is ±Inf, ±0 shall be returned, and **iptr* shall be set to ±Inf.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *modf*( ) function computes the function result and **iptr* such that:

```
a = modf(x, &iptr) ;
x == a+*iptr ;
```

allowing for the usual floating-point inaccuracies.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

frexp( ), *isnan*( ), *ldexp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *modff*( ) and *modfl*( ) functions are added for alignment with the ISO/IEC 9899: 1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

mprotect — set protection of memory mapping

SYNOPSIS

MPR

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

DESCRIPTION

The *mprotect*( ) function shall change the access protections to be that specified by *prot* for those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. The parameter *prot* determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The *prot* argument should be either PROT_NONE or the bitwise-inclusive OR of one or more of PROT_READ, PROT_WRITE, and PROT_EXEC.

If an implementation cannot support the combination of access types specified by *prot*, the call to *mprotect*( ) shall fail.

An implementation may permit accesses other than those specified by *prot*; however, no implementation shall permit a write to succeed where PROT_WRITE has not been set or shall permit any access where PROT_NONE alone has been set. Implementations shall support at least the following values of *prot*: PROT_NONE, PROT_READ, PROT_WRITE, and the bitwise-inclusive OR of PROT_READ and PROT_WRITE. If PROT_WRITE is specified, the application shall ensure that it has opened the mapped objects in the specified address range with write permission, unless MAP_PRIVATE was specified in the original mapping, regardless of whether the file descriptors used to map the objects have since been closed.

The implementation shall require that *addr* be a multiple of the page size as returned by *sysconf*( ).

The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ).

When *mprotect*( ) fails for reasons other than [EINVAL], the protections on some of the pages in the range [*addr*,*addr*+*len*) may have been changed.

RETURN VALUE

Upon successful completion, *mprotect*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *mprotect*( ) function shall fail if:

[EACCES]	The *prot* argument specifies a protection that violates the access permission the process has to the underlying memory object.
[EAGAIN]	The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping and there are insufficient memory resources to reserve for locking the private page.
[EINVAL]	The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).
[ENOMEM]	Addresses in the range [*addr*,*addr*+*len*) are invalid for the address space of a process, or specify one or more pages which are not mapped.
[ENOMEM]	The *prot* argument specifies PROT_WRITE on a MAP_PRIVATE mapping, and it would require more space than the system is able to supply for locking the private pages, if required.

[ENOTSUP] The implementation does not support the combination of accesses requested in the *prot* argument.

EXAMPLES

None.

APPLICATION USAGE

The [EINVAL] error above is marked EX because it is defined as an optional error in the POSIX Realtime Extension.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *mprotect*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is largely reworded.
- [ENOTSUP] and a second form of [ENOMEM] are added as mandatory error conditions.
- [EAGAIN] is moved from the optional to the mandatory error conditions.

Issue 6

The *mprotect*( ) function is marked as part of the Memory Protection option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size as returned by *sysconf*( ).
- The [EINVAL] error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

mq_close — close a message queue (**REALTIME**)

SYNOPSIS

MSG

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);

```

DESCRIPTION

The *mq_close*( ) function shall remove the association between the message queue descriptor, *mqdes*, and its message queue. The results of using this message queue descriptor after successful return from this *mq_close*( ), and until the return of this message queue descriptor from a subsequent *mq_open*( ), are undefined.

If the process has successfully attached a notification request to the message queue via this *mqdes*, this attachment shall be removed, and the message queue is available for another process to attach for notification.

RETURN VALUE

Upon successful completion, the *mq_close*( ) function shall return a value of zero; otherwise, the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mq_close*( ) function shall fail if:

[EBADF] The *mqdes* argument is not a valid message queue descriptor.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_close*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

NAME

mq_getattr — get message queue attributes (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);

```

DESCRIPTION

The *mqdes* argument specifies a message queue descriptor.

The *mq_getattr*( ) function shall obtain status information and attributes of the message queue and the open message queue description associated with the message queue descriptor.

The results shall be returned in the **mq_attr** structure referenced by the *mqstat* argument.

Upon return, the following members shall have the values associated with the open message queue description as set when the message queue was opened and as modified by subsequent *mq_setattr*( ) calls: *mq_flags*.

The following attributes of the message queue shall be returned as set at message queue creation: *mq_maxmsg*, *mq_msgsize*.

Upon return, the following members within the **mq_attr** structure referenced by the *mqstat* argument shall be set to the current state of the message queue:

mq_curmsgs The number of messages currently on the queue.

RETURN VALUE

Upon successful completion, the *mq_getattr*( ) function shall return zero. Otherwise, the function shall return −1 and set *errno* to indicate the error.

ERRORS

The *mq_getattr*( ) function shall fail if:

[EBADF] The *mqdes* argument is not a valid message queue descriptor.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_setattr*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_getattr*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

mq_notify — notify process that a message is available (**REALTIME**)

SYNOPSIS

MSG

```
#include <mqueue.h>

int mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

DESCRIPTION

If the argument *notification* is not NULL, this function shall register the calling process to be notified of message arrival at an empty message queue associated with the specified message queue descriptor, *mqdes*. The notification specified by the *notification* argument shall be sent to the process when the message queue transitions from empty to non-empty. At any time, only one process may be registered for notification by a message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue shall fail.

If *notification* is NULL and the process is currently registered for notification by the specified message queue, the existing registration shall be removed.

When the notification is sent to the registered process, its registration shall be removed. The message queue shall then be available for registration.

If a process has registered for notification of message arrival at a message queue and some thread is blocked in *mq_receive*( ) waiting to receive a message when a message arrives at the queue, the arriving message shall satisfy the appropriate *mq_receive*( ). The resulting behavior is as if the message queue remains empty, and no notification shall be sent.

RETURN VALUE

Upon successful completion, the *mq_notify*( ) function shall return a value of zero; otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *mq_notify*( ) function shall fail if:

[EBADF]	The *mqdes* argument is not a valid message queue descriptor.
[EBUSY]	A process is already registered for notification by the message queue.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_notify*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

mq_open — open a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

mqd_t mq_open(const char *name, int oflag, ...);
```

DESCRIPTION

The *mq_open*( ) function shall establish the connection between a process and a message queue with a message queue descriptor. It shall create an open message queue description that refers to the message queue, and a message queue descriptor that refers to that open message queue description. The message queue descriptor is used by other functions to refer to that message queue. The *name* argument points to a string naming a message queue. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *mq_open*( ) with the same value of *name* shall refer to the same message queue object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined. If the *name* argument is not the name of an existing message queue and creation is not requested, *mq_open*( ) shall fail and return an error.

A message queue descriptor may be implemented using a file descriptor, in which case applications can open up to at least {OPEN_MAX} file and message queues.

The *oflag* argument requests the desired receive and/or send access to the message queue. The requested access permission to receive messages or send messages shall be granted if the calling process would be granted read or write access, respectively, to an equivalently protected file.

The value of *oflag* is the bitwise-inclusive OR of values from the following list. Applications shall specify exactly one of the first three values (access modes) below in the value of *oflag*:

O_RDONLY	Open the message queue for receiving messages. The process can use the returned message queue descriptor with *mq_receive*( ), but not *mq_send*( ). A message queue may be open multiple times in the same or different processes for receiving messages.
O_WRONLY	Open the queue for sending messages. The process can use the returned message queue descriptor with *mq_send*( ) but not *mq_receive*( ). A message queue may be open multiple times in the same or different processes for sending messages.
O_RDWR	Open the queue for both receiving and sending messages. The process can use any of the functions allowed for O_RDONLY and O_WRONLY. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_CREAT	Create a message queue. It requires two additional arguments: *mode*, which shall be of type **mode_t**, and *attr*, which shall be a pointer to a **mq_attr** structure. If the pathname *name* has already been used to create a message queue that still exists, then this flag shall have no effect, except as noted under O_EXCL. Otherwise, a message queue shall be created without any messages in it. The user ID of the message queue shall be set to the effective user ID of the process, and the group ID of the message queue shall be set to the effective

group ID of the process. The file permission bits shall be set to the value of *mode*. When bits in *mode* other than file permission bits are set, the effect is implementation-defined. If *attr* is NULL, the message queue shall be created with implementation-defined default message queue attributes. If *attr* is non-NULL and the calling process has the appropriate privilege on *name*, the message queue *mq_maxmsg* and *mq_msgsize* attributes shall be set to the values of the corresponding members in the **mq_attr** structure referred to by *attr*. If *attr* is non-NULL, but the calling process does not have the appropriate privilege on *name*, the *mq_open*() function shall fail and return an error without creating the message queue.

O_EXCL If O_EXCL and O_CREAT are set, *mq_open*( ) shall fail if the message queue *name* exists. The check for the existence of the message queue and the creation of the message queue if it does not exist shall be atomic with respect to other threads executing *mq_open*() naming the same *name* with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NONBLOCK Determines whether a *mq_send*() or *mq_receive*() waits for resources or messages that are not currently available, or fails with *errno* set to [EAGAIN]; see *mq_send*( ) and *mq_receive*( ) for details.

The *mq_open*( ) function does not add or remove messages from the queue.

RETURN VALUE

Upon successful completion, the function shall return a message queue descriptor; otherwise, the function shall return (**mqd_t**)−1 and set *errno* to indicate the error.

ERRORS

The *mq_open*( ) function shall fail if:

[EACCES] The message queue exists and the permissions specified by *oflag* are denied, or the message queue does not exist and permission to create the message queue is denied.

[EEXIST] O_CREAT and O_EXCL are set and the named message queue already exists.

[EINTR] The *mq_open*( ) function was interrupted by a signal.

[EINVAL] The *mq_open*( ) function is not supported for the given name.

[EINVAL] O_CREAT was specified in *oflag*, the value of *attr* is not NULL, and either *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.

[EMFILE] Too many message queue descriptors or file descriptors are currently in use by this process.

[ENAMETOOLONG]
The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

[ENFILE] Too many message queues are currently open in the system.

[ENOENT] O_CREAT is not set and the named message queue does not exist.

[ENOSPC] There is insufficient space for the creation of the new message queue.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_timedreceive*( ), *mq_timedsend*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_open*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedreceive*( ) and *mq_timedsend*( ) functions are added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

NAME

mq_receive, mq_timedreceive — receive a message from a message queue (**REALTIME**)

SYNOPSIS

MSG

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
    unsigned *msg_prio);
```

MSG TMO

```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *mq_receive*( ) function shall receive the oldest of the highest priority message(s) from the message queue specified by *mqdes*. If the size of the buffer in bytes, specified by the *msg_len* argument, is less than the *mq_msgsize* attribute of the message queue, the function shall fail and return an error. Otherwise, the selected message shall be removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument.

If the value of *msg_len* is greater than {SSIZE_MAX}, the result is implementation-defined.

If the argument *msg_prio* is not NULL, the priority of the selected message shall be stored in the location referenced by *msg_prio*.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_receive*( ) shall block until a message is enqueued on the message queue or until *mq_receive*( ) is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, then the thread of highest priority that has been waiting the longest shall be selected to receive the message. Otherwise, it is unspecified which waiting thread receives the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with *mqdes*, no message shall be removed from the queue, and *mq_receive*( ) shall return an error.

TMO The *mq_timedreceive*( ) function shall receive the oldest of the highest priority messages from the message queue specified by *mqdes* as described for the *mq_receive*( ) function. However, if O_NONBLOCK was not specified when the message queue was opened via the *mq_open*( ) function, and no message exists on the queue to satisfy the receive, the wait for such a message shall be terminated when the specified timeout expires. If O_NONBLOCK is set, this function is equivalent to *mq_receive*( ).

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function.

TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The *timespec* argument is defined in the **<time.h>** header.

Under no circumstance shall the operation fail with a timeout if a message can be removed from the message queue immediately. The validity of the *abs_timeout* parameter need not be checked if a message can be removed from the message queue immediately.

RETURN VALUE

TMO Upon successful completion, the *mq_receive*( ) and *mq_timedreceive*( ) functions shall return the length of the selected message in bytes and the message shall be removed from the queue. Otherwise, no message shall be removed from the queue, the functions shall return a value of −1, and set *errno* to indicate the error.

ERRORS

TMO The *mq_receive*( ) and *mq_timedreceive*( ) functions shall fail if:

	[EAGAIN]	O_NONBLOCK was set in the message description associated with *mqdes*, and the specified message queue is empty.
	[EBADF]	The *mqdes* argument is not a valid message queue descriptor open for reading.
	[EMSGSIZE]	The specified message buffer size, *msg_len*, is less than the message size attribute of the message queue.
TMO	[EINTR]	The *mq_receive*( ) or *mq_timedreceive*( ) operation was interrupted by a signal.
TMO	[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.
TMO	[ETIMEDOUT]	The O_NONBLOCK flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

TMO The *mq_receive*( ) and *mq_timedreceive*( ) functions may fail if:

[EBADMSG]	The implementation has detected a data corruption problem with the message.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_receive*( ) function is marked as part of the Message Passing option.

The Open Group Corrigendum U021/4 is applied. The DESCRIPTION is changed to refer to *msg_len* rather than *maxsize*.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In this function it is possible for the return value to exceed the range of the type **ssize_t** (since **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of the **size_t** object is added to the description to resolve this conflict.

The *mq_timedreceive*( ) function is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *mq_timedreceive*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

IEEE PASC Interpretation 1003.1 #109 is applied, correcting the return type for *mq_timedreceive*( ) from **int** to **ssize_t**.

NAME

mq_send, mq_timedsend — send a message to a message queue (**REALTIME**)

SYNOPSIS

MSG
```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio);

```

MSG TMO
```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio, const struct timespec *abs_timeout);

```

DESCRIPTION

The *mq_send*( ) function shall add the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message, in bytes, pointed to by *msg_ptr*. The value of *msg_len* shall be less than or equal to the *mq_msgsize* attribute of the message queue, or *mq_send*( ) shall fail.

If the specified message queue is not full, *mq_send*( ) shall behave as if the message is inserted into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* shall be inserted before messages with lower values of *msg_prio*. A message shall be inserted after other messages in the queue, if any, with equal *msg_prio*. The value of *msg_prio* shall be less than {MQ_PRIO_MAX}.

If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, *mq_send*( ) shall block until space becomes available to enqueue the message, or until *mq_send*( ) is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the Priority Scheduling option is supported, then the thread of the highest priority that has been waiting the longest shall be unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and O_NONBLOCK is set in the message queue description associated with *mqdes*, the message shall not be queued and *mq_send*( ) shall return an error.

TMO The *mq_timedsend*( ) function shall add a message to the message queue specified by *mqdes* in the manner defined for the *mq_send*( ) function. However, if the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue shall be terminated when the specified timeout expires. If O_NONBLOCK is set in the message queue description, this function shall be equivalent to *mq_send*( ).

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function.

TMO The resolution of the timeout shall be the resolution of the clock on which it is based. The *timespec* argument is defined in the **<time.h>** header.

Under no circumstance shall the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the *abs_timeout* parameter need not be checked when there is sufficient room in the queue.

RETURN VALUE

TMO Upon successful completion, the *mq_send*( ) and *mq_timedsend*( ) functions shall return a value of zero. Otherwise, no message shall be enqueued, the functions shall return −1, and *errno* shall be set to indicate the error.

ERRORS

TMO The *mq_send*( ) and *mq_timedsend*( ) functions shall fail if:

	[EAGAIN]	The O_NONBLOCK flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.
	[EBADF]	The *mqdes* argument is not a valid message queue descriptor open for writing.
TMO	[EINTR]	A signal interrupted the call to *mq_send*( ) or *mq_timedsend*( ).
	[EINVAL]	The value of *msg_prio* was outside the valid range.
TMO	[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.
	[EMSGSIZE]	The specified message length, *msg_len*, exceeds the message size attribute of the message queue.
TMO	[ETIMEDOUT]	The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

EXAMPLES

None.

APPLICATION USAGE

The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the application. Message priorities range from 0 to {MQ_PRIO_MAX}−1.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_receive*( ), *mq_setattr*( ), *mq_timedreceive*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_send*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend*( ) function is added for alignment with IEEE Std 1003.1d-1999.

NAME

mq_setattr — set message queue attributes (**REALTIME**)

SYNOPSIS

MSG

```
#include <mqueue.h>

int mq_setattr(mqd_t mqdes, const struct mq_attr *restrict mqstat,
    struct mq_attr *restrict omqstat);
```

DESCRIPTION

The *mq_setattr*( ) function shall set attributes associated with the open message queue description referenced by the message queue descriptor specified by *mqdes*.

The message queue attributes corresponding to the following members defined in the **mq_attr** structure shall be set to the specified values upon successful completion of *mq_setattr*( ):

mq_flags	The value of this member is the bitwise-logical OR of zero or more of O_NONBLOCK and any implementation-defined flags.

The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* members of the **mq_attr** structure shall be ignored by *mq_setattr*( ).

If *omqstat* is non-NULL, the *mq_setattr*( ) function shall store, in the location referenced by *omqstat*, the previous message queue attributes and the current queue status. These values shall be the same as would be returned by a call to *mq_getattr*( ) at that point.

RETURN VALUE

Upon successful completion, the function shall return a value of zero and the attributes of the message queue shall have been changed as specified.

Otherwise, the message queue attributes shall be unchanged, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mq_setattr*( ) function shall fail if:

[EBADF]	The *mqdes* argument is not a valid message queue descriptor.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_open( ), *mq_send*( ), *mq_timedsend*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_setattr*( ) function is marked as part of the Message Passing option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

The *mq_timedsend*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *mq_setattr*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

NAME

mq_timedreceive — receive a message from a message queue (**ADVANCED REALTIME**)

SYNOPSIS

MSG TMO
```
#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
    size_t msg_len, unsigned *restrict msg_prio,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

Refer to *mq_receive*( ).

NAME

mq_timedsend — send a message to a message queue (**ADVANCED REALTIME**)

SYNOPSIS

MSG TMO

```
#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned msg_prio, const struct timespec *abs_timeout);
```

DESCRIPTION

Refer to *mq_send*( ).

NAME

mq_unlink — remove a message queue (**REALTIME**)

SYNOPSIS

MSG

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

DESCRIPTION

The *mq_unlink*( ) function shall remove the message queue named by the pathname *name*. After a successful call to *mq_unlink*( ) with *name*, a call to *mq_open*( ) with *name* shall fail if the flag O_CREAT is not set in *flags*. If one or more processes have the message queue open when *mq_unlink*( ) is called, destruction of the message queue shall be postponed until all references to the message queue have been closed.

Calls to *mq_open*( ) to recreate the message queue may fail until the message queue is actually removed. However, the *mq_unlink*( ) call need not block until all references have been closed; it may return immediately.

RETURN VALUE

Upon successful completion, the function shall return a value of zero. Otherwise, the named message queue shall be unchanged by this function call, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *mq_unlink*( ) function shall fail if:

[EACCES]	Permission is denied to unlink the named message queue.
[ENAMETOOLONG]	The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	The named message queue does not exist.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_open*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<mqueue.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *mq_unlink*( ) function is marked as part of the Message Passing option.

The Open Group Corrigendum U021/5 is applied, clarifying that upon unsuccessful completion, the named message queue is unchanged by this function.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Message Passing option.

NAME

mrand48 — generate uniformly distributed pseudo-random signed long integers

SYNOPSIS

XSI

```
#include <stdlib.h>

long mrand48(void);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

msgctl — XSI message control operations

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

DESCRIPTION

The *msgctl*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgctl*( ) function shall provide message control operations as specified by *cmd*. The following values for *cmd*, and the message control operations they specify, are:

IPC_STAT	Place the current value of each member of the **msqid_ds** data structure associated with *msqid* into the structure pointed to by *buf*. The contents of this structure are defined in **<sys/msg.h>**.

IPC_SET	Set the value of the following members of the **msqid_ds** data structure associated with *msqid* to the corresponding value found in the structure pointed to by *buf*:
	`msg_perm.uid`
	`msg_perm.gid`
	`msg_perm.mode`
	`msg_qbytes`
	IPC_SET can only be executed by a process with appropriate privileges or that has an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*. Only a process with appropriate privileges can raise the value of **msg_qbytes**.

IPC_RMID	Remove the message queue identifier specified by *msqid* from the system and destroy the message queue and **msqid_ds** data structure associated with it. IPC_RMD can only be executed by a process with appropriate privileges or one that has an effective user ID equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the **msqid_ds** data structure associated with *msqid*.

RETURN VALUE

Upon successful completion, *msgctl*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *msgctl*( ) function shall fail if:

[EACCES]	The argument *cmd* is IPC_STAT and the calling process does not have read permission; see Section 2.7 (on page 489).
[EINVAL]	The value of *msqid* is not a valid message queue identifier; or the value of *cmd* is not a valid command.
[EPERM]	The argument *cmd* is IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of **msg_perm.cuid** or **msg_perm.uid** in the data structure associated with *msqid*.

[EPERM]	The argument *cmd* is IPC_SET, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process does not have appropriate privileges.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgget — get the XSI message queue identifier

SYNOPSIS

XSI
```
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

DESCRIPTION

The *msgget*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgget*( ) function shall return the message queue identifier associated with the argument *key*.

A message queue identifier, associated message queue, and data structure (see **<sys/msg.h>**), shall be created for the argument *key* if one of the following is true:

- The argument *key* is equal to IPC_PRIVATE.
- The argument *key* does not already have a message queue identifier associated with it, and (*msgflg* & IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new message queue identifier shall be initialized as follows:

- **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** shall be set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of **msg_perm.mode** shall be set equal to the low-order 9 bits of *msgflg*.
- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** shall be set equal to 0.
- **msg_ctime** shall be set equal to the current time.
- **msg_qbytes** shall be set equal to the system limit.

RETURN VALUE

Upon successful completion, *msgget*( ) shall return a non-negative integer, namely a message queue identifier. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *msgget*( ) function shall fail if:

[EACCES]	A message queue identifier exists for the argument *key*, but operation permission as specified by the low-order 9 bits of *msgflg* would not be granted; see Section 2.7 (on page 489).
[EEXIST]	A message queue identifier exists for the argument *key* but ((*msgflg* & IPC_CREAT) && (*msgflg* & IPC_EXCL)) is non-zero.
[ENOENT]	A message queue identifier does not exist for the argument *key* and (*msgflg* & IPC_CREAT) is 0.
[ENOSPC]	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgctl*( ), *msgrcv*( ), *msgsnd*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

msgrcv — XSI message receive operation

SYNOPSIS

XSI

```
#include <sys/msg.h>

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
    int msgflg);
```

DESCRIPTION

The *msgrcv*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgrcv*( ) function shall read a message from the queue associated with the message queue identifier specified by *msqid* and place it in the user-defined buffer pointed to by *msgp*.

The application shall ensure that the argument *msgp* points to a user-defined buffer that contains first a field of type **long** specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long    mtype;     /* Message type. */
    char    mtext[1];  /* Message text. */
}
```

The structure member *mtype* is the received message's type as specified by the sending process.

The structure member *mtext* is the text of the message.

The argument *msgsz* specifies the size in bytes of *mtext*. The received message shall be truncated to *msgsz* bytes if it is larger than *msgsz* and (*msgflg* & MSG_NOERROR) is non-zero. The truncated part of the message shall be lost and no indication of the truncation shall be given to the calling process.

If the value of *msgsz* is greater than {SSIZE_MAX}, the result is implementation-defined.

The argument *msgtyp* specifies the type of message requested as follows:

- If *msgtyp* is 0, the first message on the queue shall be received.
- If *msgtyp* is greater than 0, the first message of type *msgtyp* shall be received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* shall be received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If (*msgflg* & IPC_NOWAIT) is non-zero, the calling thread shall return immediately with a return value of –1 and *errno* set to [ENOMSG].
- If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - — A message of the desired type is placed on the queue.
  - — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* shall be set equal to [EIDRM] and –1 shall be returned.

  - The calling thread receives a signal that is to be caught; in this case a message is not received and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*:

- **msg_qnum** shall be decremented by 1. |
- **msg_lrpid** shall be set equal to the process ID of the calling process. |
- **msg_rtime** shall be set equal to the current time. |

RETURN VALUE

Upon successful completion, *msgrcv*( ) shall return a value equal to the number of bytes actually placed into the buffer *mtext*. Otherwise, no message shall be received, *msgrcv*( ) shall return (**ssize_t**)−1, and *errno* shall be set to indicate the error.

ERRORS

The *msgrcv*( ) function shall fail if:

[E2BIG]	The value of *mtext* is greater than *msgsz* and (*msgflg* & MSG_NOERROR) is 0.
[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EIDRM]	The message queue identifier *msqid* is removed from the system.
[EINTR]	The *msgrcv*( ) function was interrupted by a signal.
[EINVAL]	*msqid* is not a valid message queue identifier.
[ENOMSG]	The queue does not contain a message of the desired type and (*msgflg* & IPC_NOWAIT) is non-zero.

EXAMPLES

Receiving a Message

The following example receives the first message on the queue (based on the value of the *msgtyp* argument, 0). The queue is identified by the *msqid* argument (assuming that the value has previously been set). This call specifies that an error should be reported if no message is available, but not if the message is too large. The message size is calculated directly using the *sizeof* operator.

```
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;
long msgtyp = 0;
...
result = msgrcv(msqid, (void *) &msg, sizeof(msg.text),
         msgtyp, MSG_NOERROR | IPC_NOWAIT);
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgsnd*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The type of the return value is changed from **int** to **ssize_t**, and a warning is added to the DESCRIPTION about values of *msgsz* larger the {SSIZE_MAX}.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

msgsnd — XSI message send operation

SYNOPSIS

XSI

```
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

DESCRIPTION

The *msgsnd*( ) function operates on XSI message queues (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.224, Message Queue). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *msgsnd*( ) function shall send a message to the queue associated with the message queue identifier specified by *msqid*.

The application shall ensure that the argument *msgp* points to a user-defined buffer that contains first a field of type **long** specifying the type of the message, and then a data portion that holds the data bytes of the message. The structure below is an example of what this user-defined buffer might look like:

```
struct mymsg {
    long   mtype;       /* Message type. */
    char   mtext[1];    /* Message text. */
}
```

The structure member *mtype* is a non-zero positive type **long** that can be used by the receiving process for message selection.

The structure member *mtext* is any text of length *msgsz* bytes. The argument *msgsz* can range from 0 to a system-imposed maximum.

The argument *msgflg* specifies the action to be taken if one or more of the following are true:

- The number of bytes already on the queue is equal to **msg_qbytes**; see **<sys/msg.h>**.
- The total number of messages on all queues system-wide is equal to the system-imposed limit.

These actions are as follows:

- If (*msgflg* & IPC_NOWAIT) is non-zero, the message shall not be sent and the calling thread shall return immediately.
- If (*msgflg* & IPC_NOWAIT) is 0, the calling thread shall suspend execution until one of the following occurs:
  - — The condition responsible for the suspension no longer exists, in which case the message is sent.
  - — The message queue identifier *msqid* is removed from the system; when this occurs, *errno* shall be set equal to [EIDRM] and −1 shall be returned.
  - — The calling thread receives a signal that is to be caught; in this case the message is not sent and the calling thread resumes execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the following actions are taken with respect to the data structure associated with *msqid*; see **<sys/msg.h>**:

- **msg_qnum** shall be incremented by 1.
- **msg_lspid** shall be set equal to the process ID of the calling process.
- **msg_stime** shall be set equal to the current time.

RETURN VALUE

Upon successful completion, *msgsnd*( ) shall return 0; otherwise, no message shall be sent, *msgsnd*( ) shall return −1, and *errno* shall be set to indicate the error.

ERRORS

The *msgsnd*( ) function shall fail if:

[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EAGAIN]	The message cannot be sent for one of the reasons cited above and (*msgflg* & IPC_NOWAIT) is non-zero.
[EIDRM]	The message queue identifier *msgid* is removed from the system.
[EINTR]	The *msgsnd*( ) function was interrupted by a signal.
[EINVAL]	The value of *msqid* is not a valid message queue identifier, or the value of *mtype* is less than 1; or the value of *msgsz* is less than 0 or greater than the system-imposed limit.

EXAMPLES

Sending a Message

The following example sends a message to the queue identified by the *msqid* argument (assuming that value has previously been set). This call specifies that an error should be reported if no message is available. The message size is calculated directly using the *sizeof* operator.

```
#include <sys/msg.h>
...
int result;
int msqid;
struct message {
    long type;
    char text[20];
} msg;

msg.type = 1;
strcpy(msg.text, "This is message 1");
...
result = msgsnd(msqid, (void *) &msg, sizeof(msg.text), IPC_NOWAIT);
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication (IPC). Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mq_close( ), *mq_getattr*( ), *mq_notify*( ), *mq_open*( ), *mq_receive*( ), *mq_send*( ), *mq_setattr*( ), *mq_unlink*( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/msg.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

msync — synchronize memory with physical storage

SYNOPSIS

MF SIO

```
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

DESCRIPTION

The *msync*( ) function shall write all modified data to permanent storage locations, if any, in those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes. If no such storage exists, *msync*( ) need not have any effect. If requested, the *msync*( ) function shall then invalidate cached copies of data. |

The implementation shall require that *addr* be a multiple of the page size as returned by *sysconf*( ).

For mappings to files, the *msync*( ) function shall ensure that all write operations are completed as defined for synchronized I/O data integrity completion. It is unspecified whether the implementation also writes out other file attributes. When the *msync*( ) function is called on MAP_PRIVATE mappings, any modified data shall not be written to the underlying object and shall not cause such data to be made visible to other processes. It is unspecified whether data in MAP_PRIVATE mappings has any permanent storage locations. SHM|TYM The effect of *msync*( ) on a shared memory object or a typed memory object is unspecified. The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ). |

The *flags* argument is constructed from the bitwise-inclusive OR of one or more of the following flags defined in the **<sys/mman.h>** header:

Symbolic Constant	Description
MS_ASYNC	Perform asynchronous writes.
MS_SYNC	Perform synchronous writes.
MS_INVALIDATE	Invalidate cached data.

When MS_ASYNC is specified, *msync*( ) shall return immediately once all the write operations are initiated or queued for servicing; when MS_SYNC is specified, *msync*( ) shall not return until all write operations are completed as defined for synchronized I/O data integrity completion. Either MS_ASYNC or MS_SYNC is specified, but not both.

When MS_INVALIDATE is specified, *msync*( ) shall invalidate all cached copies of mapped data that are inconsistent with the permanent storage locations such that subsequent references shall obtain data that was consistent with the permanent storage locations sometime between the call to *msync*( ) and the first subsequent memory reference to the data. |

If *msync*( ) causes any write to a file, the file's *st_ctime* and *st_mtime* fields shall be marked for update.

RETURN VALUE

Upon successful completion, *msync*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *msync*( ) function shall fail if:

[EBUSY]	Some or all of the addresses in the range starting at *addr* and continuing for *len* bytes are locked, and MS_INVALIDATE is specified.

[EINVAL]	The value of *flags* is invalid.
[EINVAL]	The value of *addr* is not a multiple of the page size, {PAGESIZE}.
[ENOMEM]	The addresses in the range starting at *addr* and continuing for *len* bytes are outside the range allowed for the address space of a process or specify one or more pages that are not mapped.

EXAMPLES

None.

APPLICATION USAGE

The *msync*( ) function is only supported if the Memory Mapped Files option and the Synchronized Input and Output option are supported, and thus need not be available on all implementations.

The *msync*( ) function should be used by programs that require a memory object to be in a known state; for example, in building transaction facilities.

Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that *msync*( ) is the only control over when pages are or are not written to disk.

RATIONALE

The *msync*( ) function writes out data in a mapped region to the permanent storage for the underlying object. The call to *msync*( ) ensures data integrity of the file.

After the data is written out, any cached data may be invalidated if the MS_INVALIDATE flag was specified. This is useful on systems that do not support read/write consistency.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *msync*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- [EBUSY] and a new form of [EINVAL] are added as mandatory error conditions.

Issue 6

The *msync*( ) function is marked as part of the Memory Mapped Files and Synchronized Input and Output options.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The [EBUSY] mandatory error condition is added.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- The second [EINVAL] error condition is made mandatory.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding reference to typed memory objects.

NAME

munlock — unlock a range of process address space

SYNOPSIS

MLR

```
#include <sys/mman.h>

int munlock(const void *addr, size_t len);
```

DESCRIPTION

Refer to *mlock*( ).

NAME

munlockall — unlock the address space of a process

SYNOPSIS

ML
```
#include <sys/mman.h>

int munlockall(void);
```

DESCRIPTION

Refer to *mlockall*( ).

NAME

munmap — unmap pages of memory

SYNOPSIS

MF|SHM

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

DESCRIPTION

The *munmap*( ) function shall remove any mappings for those entire pages containing any part of the address space of the process starting at *addr* and continuing for *len* bytes. Further references to these pages shall result in the generation of a SIGSEGV signal to the process. If there are no mappings in the specified address range, then *munmap*( ) has no effect. |

The implementation shall require that *addr* be a multiple of the page size {PAGESIZE}.

If a mapping to be removed was private, any modifications made in this address range shall be discarded.

ML|MLR Any memory locks (see *mlock*( ) and *mlockall*( )) associated with this address range shall be removed, as if by an appropriate call to *munlock*( ).

TYM If a mapping removed from a typed memory object causes the corresponding address range of the memory pool to be inaccessible by any process in the system except through allocatable mappings (that is, mappings of typed memory objects opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag), then that range of the memory pool shall become deallocated and may become available to satisfy future typed memory allocation requests.

A mapping removed from a typed memory object opened with the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag shall not affect in any way the availability of that typed memory for allocation.

The behavior of this function is unspecified if the mapping was not established by a call to *mmap*( ).

RETURN VALUE

Upon successful completion, *munmap*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *munmap*( ) function shall fail if:

[EINVAL]	Addresses in the range [*addr*,*addr*+*len*) are outside the valid range for the address space of a process.
[EINVAL]	The *len* argument is 0.
[EINVAL]	The *addr* argument is not a multiple of the page size as returned by *sysconf*( ).

EXAMPLES

None.

APPLICATION USAGE

The *munmap*( ) function is only supported if the Memory Mapped Files option or the Shared Memory Objects option is supported.

RATIONALE

The *munmap*( ) function corresponds to SVR4, just as the *mmap*( ) function does.

It is possible that an application has applied process memory locking to a region that contains shared memory. If this has occurred, the *munmap*( ) call ignores those locks and, if necessary, causes those locks to be removed.

FUTURE DIRECTIONS

None.

SEE ALSO

mlock( ), *mlockall*( ), *mmap*( ), *posix_typed_mem_open*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Aligned with *munmap*( ) in the POSIX Realtime Extension as follows:

- The DESCRIPTION is extensively reworded.
- The SIGBUS error is no longer permitted to be generated.

Issue 6

The *munmap*( ) function is marked as part of the Memory Mapped Files and Shared Memory Objects option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to state that implementations require *addr* to be a multiple of the page size.
- The [EINVAL] error conditions are added.

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- Semantics for typed memory objects are added to the DESCRIPTION.
- The *posix_typed_mem_open*( ) function is added to the SEE ALSO section.

NAME

nan, nanf, nanl — return quiet NaN

SYNOPSIS

```
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The function call *nan*("*n-char-sequence*") shall be equivalent to:

```
strtod("NAN(n-char-sequence)", (char **) NULL);
```

The function call *nan*(" ") shall be equivalent to:

```
strtod("NAN()", (char **) NULL)
```

If *tagp* does not point to an *n*-**char** sequence or an empty string, the function call shall be equivalent to:

```
strtod("NAN", (char **) NULL)
```

Function calls to *nanf*( ) and *nanl*( ) are equivalent to the corresponding function calls to *strtof*( ) and *strtold*( ).

RETURN VALUE

These functions shall return a quiet NaN, if available, with content indicated through *tagp*.

If the implementation does not support quiet NaNs, these functions shall return zero.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtod( ), *strtold*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

nanosleep — high resolution sleep (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);
```

DESCRIPTION

The *nanosleep*( ) function shall cause the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activity by the system. But, except for the case of being interrupted by a signal, the suspension time shall not be less than the time specified by *rqtp*, as measured by the system clock, CLOCK_REALTIME.

The use of the *nanosleep*( ) function has no effect on the action or blockage of any signal.

RETURN VALUE

If the *nanosleep*( ) function returns because the requested time has elapsed, its return value shall be zero.

If the *nanosleep*( ) function returns because it has been interrupted by a signal, it shall return a value of −1 and set *errno* to indicate the interruption. If the *rmtp* argument is non-NULL, the **timespec** structure referenced by it is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned.

If *nanosleep*( ) fails, it shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *nanosleep*( ) function shall fail if:

[EINTR]	The *nanosleep*( ) function was interrupted by a signal.
[EINVAL]	The *rqtp* argument specified a nanosecond value less than zero or greater than or equal to 1000 million.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

It is common to suspend execution of a process for an interval in order to poll the status of a non-interrupting function. A large number of actual needs can be met with a simple extension to *sleep*( ) that provides finer resolution.

In the POSIX.1-1990 standard and SVR4, it is possible to implement such a routine, but the frequency of wakeup is limited by the resolution of the *alarm*( ) and *sleep*( ) functions. In 4.3 BSD, it is possible to write such a routine using no static storage and reserving no system facilities. Although it is possible to write a function with similar functionality to *sleep*( ) using the remainder of the timers function, such a function requires the use of signals and the reservation of some signal number. This volume of IEEE Std 1003.1-200x requires that *nanosleep*( ) be non-intrusive of the signals function.

The *nanosleep*( ) function shall return a value of 0 on success and −1 on failure or if interrupted. This latter case is different from *sleep*( ). This was done because the remaining time is returned via an argument structure pointer, *rmtp*, instead of as the return value.

FUTURE DIRECTIONS

None.

SEE ALSO

sleep( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *nanosleep*( ) function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

NAME

nearbyint, nearbyintf, nearbyintl — floating-point rounding functions

SYNOPSIS

```
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, ±0 shall be returned.

If *x* is ±Inf, *x* shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *nearbyint*( ), *nearbyintf*( ), and *nearbyintl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

XSI Range Error The result would cause an overflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, nexttowardl — next representable floating-point number

SYNOPSIS

```
#include <math.h>

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall compute the next representable floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, *nextafter*( ) shall return the largest representable floating-point number less than *x*. The *nextafter*( ), *nextafterf*( ), and *nextafterl*( ) functions shall return *y* if *x* equals *y*.

The *nexttoward*( ), *nexttowardf*( ), and *nexttowardl*( ) functions shall be equivalent to the corresponding *nextafter*( ) functions, except that the second parameter shall have type **long double** and the functions shall return *y* converted to the type of the function if *x* equals *y*. |

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the next representable floating-point value following *x* in the direction of *y*.

If *x*==*y*, *y* (of the type *x*) shall be returned.

If *x* is finite and the correct function value would overflow, a range error shall occur and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*) shall be returned as appropriate for the return type of the function.

MX If *x* or *y* is NaN, a NaN shall be returned.

If *x*!=*y* and the correct function value is subnormal, zero, or underflows, a range error shall occur, and either the correct function value (if representable) or 0.0 shall be returned.

ERRORS

These functions shall fail if:

Range Error The correct value overflows

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised. |

MX Range Error The correct value is subnormal or underflows

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *nextafter*( ) function is no longer marked as an extension.

The *nextafterf*( ), *nextafterl*( ), *nexttoward*( ), *nexttowardf*( ), *nexttowardl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

nexttoward, nexttowardf, nexttowardl — next representable floating-point number

SYNOPSIS

```
#include <math.h>

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

DESCRIPTION

Refer to *nextafter*( ).

NAME

nftw — walk a file tree

SYNOPSIS

XSI

```
#include <ftw.h>

int nftw(const char *path, int (*fn)(const char *,
    const struct stat *, int, struct FTW *), int depth, int flags);
```

DESCRIPTION

The *nftw*( ) function shall recursively descend the directory hierarchy rooted in *path*. The *nftw*( ) function has a similar effect to *ftw*( ) except that it takes an additional argument *flags*, which is a bitwise-inclusive OR of zero or more of the following flags:

FTW_CHDIR	If set, *nftw*( ) shall change the current working directory to each directory as it reports files in that directory. If clear, *nftw*( ) shall not change the current working directory.
FTW_DEPTH	If set, *nftw*( ) shall report all files in a directory before reporting the directory itself. If clear, *nftw*( ) shall report any directory before reporting the files in that directory.
FTW_MOUNT	If set, *nftw*( ) shall only report files in the same file system as *path*. If clear, *nftw*( ) shall report all files encountered during the walk.
FTW_PHYS	If set, *nftw*( ) shall perform a physical walk and shall not follow symbolic links.

If FTW_PHYS is clear and FTW_DEPTH is set, *nftw*( ) shall follow links instead of reporting them, but shall not report any directory that would be a descendant of itself. If FTW_PHYS is clear and FTW_DEPTH is clear, *nftw*( ) shall follow links instead of reporting them, but shall not report the contents of any directory that would be a descendant of itself.

At each file it encounters, *nftw*( ) shall call the user-supplied function *fn* with four arguments:

- The first argument is the pathname of the object.
- The second argument is a pointer to the **stat** buffer containing information on the object.
- The third argument is an integer giving additional information. Its value is one of the following:

FTW_F	The object is a file.
FTW_D	The object is a directory.
FTW_DP	The object is a directory and subdirectories have been visited. (This condition shall only occur if the FTW_DEPTH flag is included in *flags*.)
FTW_SL	The object is a symbolic link. (This condition shall only occur if the FTW_PHYS flag is included in *flags*.)
FTW_SLN	The object is a symbolic link that does not name an existing file. (This condition shall only occur if the FTW_PHYS flag is not included in *flags*.)
FTW_DNR	The object is a directory that cannot be read. The *fn* function shall not be called for any of its descendants.
FTW_NS	The *stat*( ) function failed on the object because of lack of appropriate permission. The **stat** buffer passed to *fn* is undefined. Failure of *stat*( ) for any other reason is considered an error and *nftw*( ) shall return −1.

- The fourth argument is a pointer to an **FTW** structure. The value of **base** is the offset of the object's filename in the pathname passed as the first argument to *fn*. The value of **level** indicates depth relative to the root of the walk, where the root level is 0.

The results are unspecified if the application-supplied *fn* function does not preserve the current working directory.

The argument *depth* sets the maximum number of file descriptors that are shall be used by *nftw*( ) while traversing the file tree. At most one file descriptor shall be used for each directory level.

The *nftw*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

The *nftw*( ) function shall continue until the first of the following conditions occurs:

- An invocation of *fn* shall return a non-zero value, in which case *nftw*( ) shall return that value.
- The *nftw*( ) function detects an error other than [EACCES] (see FTW_DNR and FTW_NS above), in which case *nftw*( ) shall return −1 and set *errno* to indicate the error.
- The tree is exhausted, in which case *nftw*( ) shall return 0.

ERRORS

The *nftw*( ) function shall fail if:

[EACCES]	Search permission is denied for any component of *path* or read permission is denied for *path*, or *fn* returns −1 and does not reset *errno*.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of *path* is not a directory.
[EOVERFLOW]	A field in the **stat** structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy.

The *nftw*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
[ENFILE]	Too many files are currently open in the system.

In addition, *errno* may be set if the function pointed by *fn* causes *errno* to be set.

EXAMPLES

The following example walks the **/tmp** directory and its subdirectories, calling the *nftw*( ) function for every directory entry, to a maximum of 5 levels deep.

```
#include <ftw.h>
...
int nftwfunc(const char *, const struct stat *, int, struct FTW *);

int nftwfunc(const char *filename, const struct stat *statptr,
    int fileflags, struct FTW *pfwt)
{
    return 0;
}
...
char *startpath = "/tmp";
int depth = 5;
int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
int ret;

ret = nftw(startpath, nftwfunc, depth, flags);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *opendir*( ), *readdir*( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ftw.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the DESCRIPTION, the definition of the *depth* argument is clarified.

Issue 6

The Open Group Base Resolution bwg97-003 is applied.

The ERRORS section is updated as follows: |

- The wording of the mandatory [ELOOP] error condition is updated. |
- A second optional [ELOOP] error condition is added. |
- The [EOVERFLOW] mandatory error condition is added. |

Text is added to the DESCRIPTION to say that the *nftw*( ) function need not be reentrant and that the results are unspecified if the application-supplied *fn* function does not preserve the current working directory. |

NAME

nice — change the nice value of a process

SYNOPSIS

XSI

```
#include <unistd.h>

int nice(int incr);
```

DESCRIPTION

The *nice*( ) function shall add the value of *incr* to the nice value of the calling process. A process' nice value is a non-negative number for which a more positive value shall result in less favorable scheduling.

A maximum nice value of 2*{NZERO}−1 and a minimum nice value of 0 shall be imposed by the system. Requests for values above or below these limits shall result in the nice value being set to the corresponding limit. Only a process with appropriate privileges can lower the nice value.

PS|TPS Calling the *nice*() function has no effect on the priority of processes or threads with policy SCHED_FIFO or SCHED_RR. The effect on processes or threads with other scheduling policies is implementation-defined.

The nice value set with *nice*( ) shall be applied to the process. If the process is multi-threaded, the nice value shall affect all system scope threads in the process.

As −1 is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *nice*( ), and if it returns −1, check to see whether *errno* is non-zero.

RETURN VALUE

Upon successful completion, *nice*( ) shall return the new nice value −{NZERO}. Otherwise, −1 shall be returned, the process' nice value shall not be changed, and *errno* shall be set to indicate the error.

ERRORS

The *nice*( ) function shall fail if:

[EPERM] The *incr* argument is negative and the calling process does not have appropriate privileges.

EXAMPLES

Changing the Nice Value

The following example adds the value of the *incr* argument, −20, to the nice value of the calling process.

```
#include <unistd.h>
...
int incr = -20;
int ret;

ret = nice(incr);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getpriority( ), *setpriority*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A statement is added to the description indicating the effects of this function on the different scheduling policies and multi-threaded processes.

NAME

nl_langinfo — language information

SYNOPSIS

XSI
```
#include <langinfo.h>

char *nl_langinfo(nl_item item);

```

DESCRIPTION

The *nl_langinfo*( ) function shall return a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale (see **<langinfo.h>**). The manifest constant names and values of *item* are defined in **<langinfo.h>**. For example:

```
nl_langinfo(ABDAY_1)
```

would return a pointer to the string `"Dom"` if the identified language was Portuguese, and `"Sun"` if the identified language was English.

Calls to *setlocale*( ) with a category corresponding to the category of *item* (see **<langinfo.h>**), or to the category *LC_ALL*, may overwrite the array pointed to by the return value.

The *nl_langinfo*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

In a locale where *langinfo* data is not defined, *nl_langinfo*( ) shall return a pointer to the corresponding string in the POSIX locale. In all locales, *nl_langinfo*( ) shall return a pointer to an empty string if *item* contains an invalid setting.

This pointer may point to static data that may be overwritten on the next call.

ERRORS

No errors are defined.

EXAMPLES

Getting Date and Time Formatting Information

The following example returns a pointer to a string containing date and time formatting information, as defined in the *LC_TIME* category of the current locale.

```
#include <time.h>
#include <langinfo.h>
...
strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);
...
```

APPLICATION USAGE

The array pointed to by the return value should not be modified by the program, but may be modified by further calls to *nl_langinfo*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<nl_types.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 2.

Issue 5

The last paragraph of the DESCRIPTION is moved from the APPLICATION USAGE section.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

nrand48 — generate uniformly distributed pseudo-random non-negative long integers

SYNOPSIS

XSI

```
#include <stdlib.h>

long nrand48(unsigned short xsubi[3]);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

ntohl — convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

uint32_t ntohl(uint32_t netlong);
```

DESCRIPTION

Refer to *htonl*( ).

NAME

ntohs — convert values between host and network byte order

SYNOPSIS

```
#include <arpa/inet.h>

uint16_t ntohs(uint16_t netshort);
```

DESCRIPTION

Refer to *htonl*( ).

NAME

open — open a file

SYNOPSIS

OH

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ... );
```

DESCRIPTION

The *open*( ) function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The *path* argument points to a pathname naming the file.

The *open*( ) function shall return a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other process in the system. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The file offset used to mark the current position within the file shall be set to the beginning of the file.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**. Applications shall specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

Any combination of the following may be used:

O_APPEND	If set, the file offset shall be set to the end of the file prior to each write.
O_CREAT	If the file exists, this flag has no effect except as noted under O_EXCL below. Otherwise, the file shall be created; the user ID of the file shall be set to the effective user ID of the process; the group ID of the file shall be set to the group ID of the file's parent directory or to the effective group ID of the process; and the access permission bits (see **<sys/stat.h>**) of the file mode shall be set to the value of the third argument taken as type **mode_t** modified as follows: a bitwise AND is performed on the file-mode bits and the corresponding bits in the complement of the process' file mode creation mask. Thus, all bits in the file mode whose corresponding bit in the file mode creation mask is set are cleared. When bits other than the file permission bits are set, the effect is unspecified. The third argument does not affect whether the file is open for reading, writing, or for both. Implementations shall provide a way to initialize the file's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the file's group ID to the effective group ID of the calling process.
SIO O_DSYNC	Write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

O_EXCL If O_CREAT and O_EXCL are set, *open*( ) shall fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist shall be atomic with respect to other threads executing *open*( ) naming the same filename in the same directory with O_EXCL and O_CREAT set. If O_EXCL and O_CREAT are set, and *path* names a symbolic link, *open*( ) shall fail and set *errno* to [EEXIST], regardless of the contents of the symbolic link. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_NOCTTY If set and *path* identifies a terminal device, *open*( ) shall not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK When opening a FIFO with O_RDONLY or O_WRONLY set:

- If O_NONBLOCK is set, an *open*( ) for reading-only shall return without delay. An *open*( ) for writing-only shall return an error if no process currently has the file open for reading.
- If O_NONBLOCK is clear, an *open*( ) for reading-only shall block the calling thread until a thread opens the file for writing. An *open*( ) for writing-only shall block the calling thread until a thread opens the file for reading.

When opening a block special or character special file that supports non-blocking opens:

- If O_NONBLOCK is set, the *open*( ) function shall return without blocking for the device to be ready or available. Subsequent behavior of the device is device-specific.
- If O_NONBLOCK is clear, the *open*( ) function shall block the calling thread until the device is ready or available before returning.

Otherwise, the behavior of O_NONBLOCK is unspecified.

SIO O_RSYNC Read I/O operations on the file descriptor shall complete at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC are set in *oflag*, all I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in flags, all I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

SIO O_SYNC Write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

O_TRUNC If the file exists and is a regular file, and the file is successfully opened O_RDWR or O_WRONLY, its length shall be truncated to 0, and the mode and owner shall be unchanged. It shall have no effect on FIFO special files or terminal device files. Its effect on other file types is implementation-defined. The result of using O_TRUNC with O_RDONLY is undefined.

If O_CREAT is set and the file did not previously exist, upon successful completion, *open*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the file and the *st_ctime* and *st_mtime* fields of the parent directory.

If O_TRUNC is set and the file did previously exist, upon successful completion, *open*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file.

SIO If both the O_SYNC and O_DSYNC flags are set, the effect is as if only the O_SYNC flag was set.

XSR If *path* refers to a STREAMS file, *oflag* may be constructed from O_NONBLOCK OR'ed with either O_RDONLY, O_WRONLY, or O_RDWR. Other flag values are not applicable to STREAMS devices and shall have no effect on them. The value O_NONBLOCK affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK is device-specific.

XSI If *path* names the master side of a pseudo-terminal device, then it is unspecified whether *open*( ) locks the slave side so that it cannot be opened. Conforming applications shall call *unlockpt*( ) before opening the slave side.

The largest value that can be represented correctly in an object of type **off_t** shall be established as the offset maximum in the open file description.

RETURN VALUE

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, −1 shall be returned and *errno* set to indicate the error. No files shall be created or modified if the function returns −1.

ERRORS

The *open*( ) function shall fail if:

	[EACCES]	Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by *oflag* are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created, or O_TRUNC is specified and write permission is denied.
	[EEXIST]	O_CREAT and O_EXCL are set, and the named file exists.
	[EINTR]	A signal was caught during *open*( ).
SIO	[EINVAL]	The implementation does not support synchronized I/O for this file.
XSR	[EIO]	The *path* argument names a STREAMS file and a hangup or error occurred during the *open*( ).
	[EISDIR]	The named file is a directory and *oflag* includes O_WRONLY or O_RDWR.
	[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
	[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
	[ENFILE]	The maximum allowable number of files is currently open in the system.
	[ENOENT]	O_CREAT is not set and the named file does not exist; or O_CREAT is set and either the path prefix does not exist or the *path* argument points to an empty string.
XSR	[ENOSR]	The *path* argument names a STREAMS-based file and the system is unable to allocate a STREAM.
	[ENOSPC]	The directory or file system that would contain the new file cannot be expanded, the file does not exist, and O_CREAT is specified.

[ENOTDIR]	A component of the path prefix is not a directory.
[ENXIO]	O_NONBLOCK is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[EOVERFLOW]	The named file is a regular file and the size of the file cannot be represented correctly in an object of type **off_t**.
[EROFS]	The named file resides on a read-only file system and either O_WRONLY, O_RDWR, O_CREAT (if file does not exist), or O_TRUNC is set in the *oflag* argument.

The *open*( ) function may fail if:

XSI [EAGAIN]	The *path* argument names the slave side of a pseudo-terminal device that is locked.
[EINVAL]	The value of the *oflag* argument is not valid.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
XSR [ENOMEM]	The *path* argument names a STREAMS file and the system is unable to allocate resources.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed and *oflag* is O_WRONLY or O_RDWR.

EXAMPLES

Opening a File for Writing by the Owner

The following example opens the file **/tmp/file**, either by creating it (if it does not already exist), or by truncating its length to 0 (if it does exist). In the former case, if the call creates a new file, the access permission bits in the file mode of the file are set to permit reading and writing by the owner, and to permit reading only by group members and others.

If the call to *open*( ) is successful, the file is opened for writing.

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

Opening a File Using an Existence Check

The following example uses the *open*( ) function to try to create the **LOCKFILE** file and open it for writing. Since the *open*( ) function specifies the O_EXCL flag, the call fails if the file already exists. In that case, the program assumes that someone else is updating the password file and exits.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* Integer for file descriptor returned by open() call. */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}
...
```

Opening a File for Writing

The following example opens a file for writing, creating the file if it does not already exist. If the file does exist, the system truncates the file to zero bytes.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
char filename[PATH_MAX+1];
...
if ((pfd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n"); exit(1);
}
...
```

APPLICATION USAGE

None.

RATIONALE

Except as specified in this volume of IEEE Std 1003.1-200x, the flags allowed in *oflag* are not mutually-exclusive and any number of them may be used simultaneously.

Some implementations permit opening FIFOs with O_RDWR. Since FIFOs could be implemented in other ways, and since two file descriptors can be used to the same effect, this possibility is left as undefined.

See *getgroups*( ) about the group of a newly created file.

The use of *open*( ) to create a regular file is preferable to the use of *creat*( ), because the latter is redundant and included only for historical reasons.

The use of the O_TRUNC flag on FIFOs and directories (pipes cannot be *open*( )-ed) must be permissible without unexpected side effects (for example, *creat*( ) on a FIFO must not remove data). Since terminal special files might have type-ahead data stored in the buffer, O_TRUNC should not affect their content, particularly if a program that normally opens a regular file should open the current controlling terminal instead. Other file types, particularly implementation-defined ones, are left implementation-defined.

IEEE Std 1003.1-200x permits [EACCES] to be returned for conditions other than those explicitly listed.

The O_NOCTTY flag was added to allow applications to avoid unintentionally acquiring a controlling terminal as a side effect of opening a terminal file. This volume of IEEE Std 1003.1-200x does not specify how a controlling terminal is acquired, but it allows an implementation to provide this on *open*( ) if the O_NOCTTY flag is not set and other conditions specified in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface are met. The O_NOCTTY flag is an effective no-op if the file being opened is not a terminal device.

In historical implementations the value of O_RDONLY is zero. Because of that, it is not possible to detect the presence of O_RDONLY and another option. Future implementations should encode O_RDONLY and O_WRONLY as bit flags so that:

```
O_RDONLY | O_WRONLY == O_RDWR
```

In general, the *open*( ) function follows the symbolic link if *path* names a symbolic link. However, the *open*( ) function, when called with O_CREAT and O_EXCL, is required to fail with [EEXIST] if *path* names an existing symbolic link, even if the symbolic link refers to a nonexistent file. This behavior is required so that privileged applications can create a new file in a known location without the possibility that a symbolic link might cause the file to be created in a different location.

For example, a privileged application that must create a file with a predictable name in a user-writable directory, such as the user's home directory, could be compromised if the user creates a symbolic link with that name that refers to a nonexistent file in a system directory. If the user can influence the contents of a file, the user could compromise the system by creating a new system configuration or spool file that would then be interpreted by the system. The test for a symbolic link which refers to a nonexisting file must be atomic with the creation of a new file.

The POSIX.1-1990 standard required that the group ID of a newly created file be set to the group ID of its parent directory or to the effective group ID of the creating process. FIPS 151-2 required that implementations provide a way to have the group ID be set to the group ID of the containing directory, but did not prohibit implementations also supporting a way to set the group ID to the effective group ID of the creating process. Conforming applications should not assume which group ID will be used. If it matters, an application can use *chown*( ) to set the group ID after the file is created, or determine under what conditions the implementation will set the desired group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *close*( ), *creat*( ), *dup*( ), *fcntl*( ), *lseek*( ), *read*( ), *umask*( ), *unlockpt*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION, O_CREAT is amended to state that the group ID of the file is set to the group ID of the file's parent directory or to the effective group ID of the process. This is a FIPS requirement.
- In the DESCRIPTION, text is added to indicate setting of the offset maximum in the open file description. This change is to support large files.
- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [ENXIO] mandatory error condition is added.
- The [EINVAL], [ENAMETOOLONG], and [ETXTBSY] optional error conditions are added.

The DESCRIPTION and ERRORS sections are updated so that items related to the optional XSI STREAMS Option Group are marked.

The following changes were made to align with the IEEE P1003.1a draft standard:

- An explanation is added of the effect of the O_CREAT and O_EXCL flags when the path refers to a symbolic link.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The DESCRIPTION of O_EXCL is updated in response to IEEE PASC Interpretation 1003.1c #48.

NAME

opendir — open a directory

SYNOPSIS

```
#include <dirent.h>

DIR *opendir(const char *dirname);
```

DESCRIPTION

The *opendir*( ) function shall open a directory stream corresponding to the directory named by the *dirname* argument. The directory stream is positioned at the first entry. If the type **DIR** is implemented using a file descriptor, applications shall only be able to open up to a total of {OPEN_MAX} files and directories.

RETURN VALUE

Upon successful completion, *opendir*( ) shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

ERRORS

The *opendir*( ) function shall fail if:

[EACCES]	Search permission is denied for the component of the path prefix of *dirname* or read permission is denied for *dirname*.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *dirname* argument.
[ENAMETOOLONG]	The length of the *dirname* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *dirname* does not name an existing directory or *dirname* is an empty string.
[ENOTDIR]	A component of *dirname* is not a directory.

The *opendir*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *dirname* argument.
[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *dirname* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
[ENFILE]	Too many files are currently open in the system.

EXAMPLES

Open a Directory Stream

The following program fragment demonstrates how the *opendir*( ) function is used.

```
#include <sys/types.h>
#include <dirent.h>
#include <libgen.h>
...
    DIR *dir;
    struct dirent *dp;
...
    if ((dir = opendir (".")) == NULL) {
        perror ("Cannot open .");
        exit (1);
    }

    while ((dp = readdir (dir)) != NULL) {
...
```

APPLICATION USAGE

The *opendir*( ) function should be used in conjunction with *readdir*( ), *closedir*( ), and *rewinddir*( ) to examine the contents of the directory (see the EXAMPLES section in *readdir*( )). This method is recommended for portability.

RATIONALE

Based on historical implementations, the rules about file descriptors apply to directory streams as well. However, this volume of IEEE Std 1003.1-200x does not mandate that the directory stream be implemented using file descriptors. The description of *closedir*( ) clarifies that if a file descriptor is used for the directory stream, it is mandatory that *closedir*( ) deallocate the file descriptor. When a file descriptor is used to implement the directory stream, it behaves as if the FD_CLOEXEC had been set for the file descriptor.

The directory entries for dot and dot-dot are optional. This volume of IEEE Std 1003.1-200x does not provide a way to test *a priori* for their existence because an application that is portable must be written to look for (and usually ignore) those entries. Writing code that presumes that they are the first two entries does not always work, as many implementations permit them to be other than the first two entries, with a ''normal'' entry preceding them. There is negligible value in providing a way to determine what the implementation does because the code to deal with dot and dot-dot must be written in any case and because such a flag would add to the list of those flags (which has proven in itself to be objectionable) and might be abused.

Since the structure and buffer allocation, if any, for directory operations are defined by the implementation, this volume of IEEE Std 1003.1-200x imposes no portability requirements for erroneous program constructs, erroneous data, or the use of unspecified values such as the use or referencing of a *dirp* value or a **dirent** structure value after a directory stream has been closed or after a *fork*( ) or one of the *exec* function calls.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *lstat*( ), *readdir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**, **<limits.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

openlog — open a connection to the logging facility

SYNOPSIS

XSI

```
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
```

DESCRIPTION

Refer to *closelog*( ).

NAME

optarg, opterr, optind, optopt — options parsing variables

SYNOPSIS

```
#include <unistd.h>

extern char *optarg;
extern int opterr, optind, optopt;
```

DESCRIPTION

Refer to *getopt*( ).

NAME

pathconf — get configurable pathname variables

SYNOPSIS

```
#include <unistd.h>

long pathconf(const char *path, int name);
```

DESCRIPTION

Refer to *fpathconf*( ).

NAME

pause — suspend the thread until a signal is received

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

DESCRIPTION

The *pause*( ) function shall suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

If the action is to terminate the process, *pause*( ) shall not return.

If the action is to execute a signal-catching function, *pause*( ) shall return after the signal-catching function returns.

RETURN VALUE

Since *pause*( ) suspends thread execution indefinitely unless interrupted by a signal, there is no successful completion return value. A value of −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *pause*( ) function shall fail if:

[EINTR]	A signal is caught by the calling process and control is returned from the signal-catching function.

EXAMPLES

None.

APPLICATION USAGE

Many common uses of *pause*( ) have timing windows. The scenario involves checking a condition related to a signal and, if the signal has not occurred, calling *pause*( ). When the signal occurs between the check and the call to *pause*( ), the process often blocks indefinitely. The *sigprocmask*( ) and *sigsuspend*( ) functions can be used to avoid this type of problem.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sigsuspend( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The APPLICATION USAGE section is added.

NAME

pclose — close a pipe stream to or from a process

SYNOPSIS

CX

```
#include <stdio.h>

int pclose(FILE *stream);
```

DESCRIPTION

The *pclose*( ) function shall close a stream that was opened by *popen*( ), wait for the command to terminate, and return the termination status of the process that was running the command language interpreter. However, if a call caused the termination status to be unavailable to *pclose*( ), then *pclose*( ) shall return –1 with *errno* set to [ECHILD] to report this situation. This can happen if the application calls one of the following functions:

- *wait*( )
- *waitpid*( ) with a *pid* argument less than or equal to 0 or equal to the process ID of the command line interpreter
- Any other function not defined in this volume of IEEE Std 1003.1-200x that could do one of the above

In any case, *pclose*( ) shall not return before the child process created by *popen*( ) has terminated.

If the command language interpreter cannot be executed, the child termination status returned by *pclose*( ) shall be as if the command language interpreter terminated using *exit*(127) or *_exit*(127).

The *pclose*( ) function shall not affect the termination status of any child of the calling process other than the one created by *popen*( ) for the associated stream.

If the argument *stream* to *pclose*( ) is not a pointer to a stream created by *popen*( ), the result of *pclose*( ) is undefined.

RETURN VALUE

Upon successful return, *pclose*( ) shall return the termination status of the command language interpreter. Otherwise, *pclose*( ) shall return –1 and set *errno* to indicate the error.

ERRORS

The *pclose*( ) function shall fail if:

[ECHILD]	The status of the child process could not be obtained, as described above.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

There is a requirement that *pclose*( ) not return before the child process terminates. This is intended to disallow implementations that return [EINTR] if a signal is received while waiting. If *pclose*( ) returned before the child terminated, there would be no way for the application to discover which child used to be associated with the stream, and it could not do the cleanup itself.

If the stream pointed to by *stream* was not created by *popen*( ), historical implementations of *pclose*( ) return –1 without setting *errno*. To avoid requiring *pclose*( ) to set *errno* in this case, IEEE Std 1003.1-200x makes the behavior unspecified. An application should not use *pclose*( ) to close any stream that was not created by *popen*( ).

Some historical implementations of *pclose*( ) either block or ignore the signals SIGINT, SIGQUIT, and SIGHUP while waiting for the child process to terminate. Since this behavior is not described for the *pclose*( ) function in IEEE Std 1003.1-200x, such implementations are not conforming. Also, some historical implementations return [EINTR] if a signal is received, even though the child process has not terminated. Such implementations are also considered non-conforming.

Consider, for example, an application that uses:

```
popen("command", "r")
```

to start *command*, which is part of the same application. The parent writes a prompt to its standard output (presumably the terminal) and then reads from the stream. The child reads the response from the user, does some transformation on the response (pathname expansion, perhaps) and writes the result to its standard output. The parent process reads the result from the pipe, does something with it, and prints another prompt. The cycle repeats. Assuming that both processes do appropriate buffer flushing, this would be expected to work.

To conform to IEEE Std 1003.1-200x, *pclose*( ) must use *waitpid*( ), or some similar function, instead of *wait*( ).

The code sample below illustrates how the *pclose*( ) function might be implemented on a system conforming to IEEE Std 1003.1-200x.

```
int pclose(FILE *stream)
{
    int stat;
    pid_t pid;

    pid = <pid for process created for stream by popen()>
    (void) fclose(stream);
    while (waitpid(pid, &stat, 0) == -1) {
        if (errno != EINTR){
            stat = -1;
            break;
        }
    }
    return(stat);
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *popen*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

perror — write error messages to standard error

SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *perror*( ) function shall map the error number accessed through the symbol *errno* to a language-dependent error message, which shall be written to the standard error stream as follows: |

- First (if *s* is not a null pointer and the character pointed to by *s* is not the null byte), the string pointed to by *s* followed by a colon and a <space>.
- Then an error message string followed by a <newline>.

The contents of the error message strings shall be the same as those returned by *strerror*( ) with argument *errno*. |

CX The *perror*( ) function shall mark the file associated with the standard error stream as having been written (*st_ctime*, *st_mtime* marked for update) at some time between its successful completion and *exit*( ), *abort*( ), or the completion of *fflush*( ) or *fclose*( ) on *stderr*.

The *perror*( ) function shall not change the orientation of the standard error stream.

RETURN VALUE

The *perror*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Printing an Error Message for a Function

The following example replaces *bufptr* with a buffer that is the necessary size. If an error occurs, the *perror*( ) function prints a message and the program exits.

```
#include <stdio.h>
#include <stdlib.h>
...
char *bufptr;
size_t szbuf;
...
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc"); exit(2);
}
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strerror( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A paragraph is added to the DESCRIPTION indicating that *perror*( ) does not change the orientation of the standard error stream.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

pipe — create an interprocess channel

SYNOPSIS

```
#include <unistd.h>

int pipe(int fildes[2]);
```

DESCRIPTION

The *pipe*( ) function shall create a pipe and place two file descriptors, one each into the arguments *fildes*[0] and *fildes*[1], that refer to the open file descriptions for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*( ) call. The O_NONBLOCK and FD_CLOEXEC flags shall be clear on both file descriptors. (The *fcntl*( ) function can be used to set both these flags.)

Data can be written to the file descriptor *fildes*[1] and read from the file descriptor *fildes*[0]. A read on the file descriptor *fildes*[0] shall access data written to the file descriptor *fildes*[1] on a first-in-first-out basis. It is unspecified whether *fildes*[0] is also open for writing and whether *fildes*[1] is also open for reading.

A process has the pipe open for reading (correspondingly writing) if it has a file descriptor open that refers to the read end, *fildes*[0] (write end, *fildes*[1]).

Upon successful completion, *pipe*( ) shall mark for update the *st_atime*, *st_ctime*, and *st_mtime* fields of the pipe.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *pipe*( ) function shall fail if:

[EMFILE]	More than {OPEN_MAX} minus two file descriptors are already in use by this process.
[ENFILE]	The number of simultaneously open files in the system would exceed a system-imposed limit.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The wording carefully avoids using the verb ‘‘to open’’ in order to avoid any implication of use of *open*( ); see also *write*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION is updated to indicate that certain dispositions of *fildes*[0] and *fildes*[1] are unspecified.

NAME

poll — input/output multiplexing

SYNOPSIS

XSI
```
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nfds, int timeout);
```

DESCRIPTION

The *poll*( ) function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. For each member of the array pointed to by *fds*, *poll*( ) shall examine the given file descriptor for the event(s) specified in *events*. The number of **pollfd** structures in the *fds* array is specified by *nfds*. The *poll*( ) function shall identify those file descriptors on which an application can read or write data, or on which certain events have occurred.

The *fds* argument specifies the file descriptors to be examined and the events of interest for each file descriptor. It is a pointer to an array with one member for each open file descriptor of interest. The array's members are **pollfd** structures within which *fd* specifies an open file descriptor and *events* and *revents* are bitmasks constructed by OR'ing a combination of the following event flags:

POLLIN	Data other than high-priority data may be read without blocking. XSR For STREAMS, this flag is set in *revents* even if the message is of zero length. This flag shall be equivalent to POLLRDNORM | POLLRDBAND.
POLLRDNORM	Normal data may be read without blocking. XSR For STREAMS, data on priority band 0 may be read without blocking. This flag is set in *revents* even if the message is of zero length.
POLLRDBAND	Priority data may be read without blocking. XSR For STREAMS, data on priority bands greater than 0 may be read without blocking. This flag is set in *revents* even if the message is of zero length.
POLLPRI	High-priority data may be read without blocking. XSR For STREAMS, this flag is set in *revents* even if the message is of zero length.
POLLOUT	Normal data may be written without blocking. XSR For STREAMS, data on priority band 0 may be written without blocking.
POLLWRNORM	Equivalent to POLLOUT.
POLLWRBAND	Priority data may be written. XSR For STREAMS, data on priority bands greater than 0 may be written without blocking. If any priority band has been written to on this STREAM, this event only examines bands that have been written to at least once.
POLLERR	An error has occurred on the device or stream. This flag is only valid in the *revents* bitmask; it shall be ignored in the *events* member.
POLLHUP	The device has been disconnected. This event and POLLOUT are mutually-exclusive; a stream can never be writable if a hangup has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND, or POLLPRI are not mutually-exclusive. This flag is only valid in the *revents* bitmask; it shall be ignored in the *events* member.

POLLNVAL	The specified *fd* value is invalid. This flag is only valid in the *revents* member; it shall ignored in the *events* member.

The significance and semantics of normal, priority, and high-priority data are file and device-specific.

If the value of *fd* is less than 0, *events* shall be ignored, and *revents* shall be set to 0 in that entry on return from *poll*( ).

In each **pollfd** structure, *poll*( ) shall clear the *revents* member, except that where the application requested a report on a condition by setting one of the bits of *events* listed above, *poll*( ) shall set the corresponding bit in *revents* if the requested condition is true. In addition, *poll*( ) shall set the POLLHUP, POLLERR, and POLLNVAL flag in *revents* if the condition is true, even if the application did not set the corresponding bit in *events*. |

If none of the defined events have occurred on any selected file descriptor, *poll*( ) shall wait at least *timeout* milliseconds for an event to occur on any of the selected file descriptors. If the value of *timeout* is 0, *poll*( ) shall return immediately. If the value of *timeout* is −1, *poll*( ) shall block until a requested event occurs or until the call is interrupted.

Implementations may place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

The *poll*( ) function shall not be affected by the O_NONBLOCK flag.

XSR The *poll*( ) function shall support regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *poll*( ) on elements of *fds* that refer to other types of file is unspecified.

Regular files shall always poll TRUE for reading and writing.

A file descriptor for a socket that is listening for connections shall indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously shall indicate that it is ready for writing, once a connection has been established.

RETURN VALUE

Upon successful completion, *poll*( ) shall return a non-negative value. A positive value indicates the total number of file descriptors that have been selected (that is, file descriptors for which the *revents* member is non-zero). A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, *poll*( ) shall return −1 and set *errno* to indicate the error.

ERRORS

The *poll*( ) function shall fail if:

[EAGAIN]	The allocation of internal data structures failed but a subsequent request may succeed.
[EINTR]	A signal was caught during *poll*( ).
XSR [EINVAL]	The *nfds* argument is greater than {OPEN_MAX}, or one of the *fd* members refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

EXAMPLES

Checking for Events on a Stream

The following example opens a pair of STREAMS devices and then waits for either one to become writable. This example proceeds as follows:

1. Sets the *timeout* parameter to 500 milliseconds.
2. Opens the STREAMS devices **/dev/dev0** and **/dev/dev1**, and then polls them, specifying POLLOUT and POLLWRBAND as the events of interest.

   The STREAMS device names **/dev/dev0** and **/dev/dev1** are only examples of how STREAMS devices can be named; STREAMS naming conventions may vary among systems conforming to the IEEE Std 1003.1-200x.
3. Uses the *ret* variable to determine whether an event has occurred on either of the two STREAMS. The *poll*( ) function is given 500 milliseconds to wait for an event to occur (if it has not occurred prior to the *poll*( ) call).
4. Checks the returned value of *ret*. If a positive value is returned, one of the following can be done:
   a. Priority data can be written to the open STREAM on priority bands greater than 0, because the POLLWRBAND event occurred on the open STREAM (*fds*[0] or *fds*[1]).
   b. Data can be written to the open STREAM on priority-band 0, because the POLLOUT event occurred on the open STREAM (*fds*[0] or *fds*[1]).
5. If the returned value is not a positive value, permission to write data to the open STREAM (on any priority band) is denied.
6. If the POLLHUP event occurs on the open STREAM (*fds*[0] or *fds*[1]), the device on the open STREAM has disconnected.

```
#include <stropts.h>
#include <poll.h>
...
struct pollfd fds[2];
int timeout_msecs = 500;
int ret;
    int i;

/* Open STREAMS device. */
fds[0].fd = open("/dev/dev0", ...);
fds[1].fd = open("/dev/dev1", ...);
    fds[0].events = POLLOUT | POLLWRBAND;
    fds[1].events = POLLOUT | POLLWRBAND;

ret = poll(fds, 2, timeout_msecs);

if (ret > 0) {
    /* An event on one of the fds has occurred. */
    for (i=0; i<2; i++) {
        if (fds[i].revents & POLLWRBAND) {
        /* Priority data may be written on device number i. */
...
        }
        if (fds[i].revents & POLLOUT) {
```

```
                /* Data may be written on device number i. */
        ...
                }
                if (fds[i].revents & POLLHUP) {
                /* A hangup has occurred on device number i. */
        ...
                }
            }
        }
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getmsg( ), *putmsg*( ), *read*( ), *select*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<poll.h>**, **<stropts.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The description of POLLWRBAND is updated.

Issue 6

Text referring to sockets is added to the DESCRIPTION.

Text relating to the XSI STREAMS Option Group is marked.

The Open Group Corrigendum Unnn/nn is applied, updating the DESCRIPTION of POLLWRBAND.

NAME

popen — initiate pipe streams to or from a process

SYNOPSIS

CX
```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

DESCRIPTION

The *popen*( ) function shall execute the command specified by the string *command*. It shall create a pipe between the calling program and the executed command, and shall return a pointer to a stream that can be used to either read from or write to the pipe.

The environment of the executed command shall be as if a child process were created within the *popen*( ) call using the *fork*( ) function, and the child invoked the *sh* utility using the call:

```
execl(shell path, "sh", "-c", command, (char *)0);
```

where *shell path* is an unspecified pathname for the *sh* utility.

The *popen*( ) function shall ensure that any streams from previous *popen*( ) calls that remain open in the parent process are closed in the new child process.

The *mode* argument to *popen*( ) is a string that specifies I/O mode:

1. If *mode* is *r*, when the child process is started, its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*( ), shall be the readable end of the pipe.
2. If *mode* is *w*, when the child process is started its file descriptor STDIN_FILENO shall be the readable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*( ), shall be the writable end of the pipe.
3. If *mode* is any other value, the result is undefined.

After *popen*( ), both the parent and the child process shall be capable of executing independently before either terminates.

Pipe streams are byte-oriented.

RETURN VALUE

Upon successful completion, *popen*( ) shall return a pointer to an open stream that can be used to read or write to the pipe. Otherwise, it shall return a null pointer and may set *errno* to indicate the error.

ERRORS

The *popen*( ) function may fail if:

[EMFILE]	{FOPEN_MAX} or {STREAM_MAX} streams are currently open in the calling process.
[EINVAL]	The *mode* argument is invalid.

The *popen*( ) function may also set *errno* values as described by *fork*( ) or *pipe*( ).

EXAMPLES

None.

APPLICATION USAGE

Since open files are shared, a mode *r* command can be used as an input filter and a mode *w* command as an output filter.

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be prevented by careful buffer flushing; for example, with *fflush*( ).

A stream opened by *popen*( ) should be closed by *pclose*( ).

The behavior of *popen*( ) is specified for values of *mode* of *r* and *w*. Other modes such as *rb* and *wb* might be supported by specific implementations, but these would not be portable features. Note that historical implementations of *popen*( ) only check to see if the first character of *mode* is *r*. Thus, a *mode* of *robert the robot* would be treated as *mode r*, and a *mode* of *anything else* would be treated as *mode w*.

If the application calls *waitpid*( ) or *waitid*( ) with a *pid* argument greater than 0, and it still has a stream that was called with *popen*( ) open, it must ensure that *pid* does not refer to the process started by *popen*( ).

To determine whether or not the environment specified in the Shell and Utilities volume of IEEE Std 1003.1-200x is present, use the function call:

```
sysconf(_SC_2_VERSION)
```

(See *sysconf*( )).

RATIONALE

The *popen*( ) function should not be used by programs that have set user (or group) ID privileges. The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used instead. This prevents any unforeseen manipulation of the environment of the user that could cause execution of commands not anticipated by the calling program.

If the original and *popen*( )ed processes both intend to read or write or read and write a common file, and either will be using FILE-type C functions (*fread*( ), *fwrite*( ), and so on), the rules for sharing file handles must be observed (see Section 2.5.1 (on page 485)).

Since open files are shared, a mode *r* argument can be used as an input filter and a mode *w* argument as an output filter.

The behavior of *popen*( ) is specified for modes of *r* and *w*. Other modes such as *rb* and *wb* might be supported by specific implementations, but these would not be portable features. Note that historical implementations of *popen*( ) only check to see if the first character of mode is `'r'`. Thus, a mode of *robert the robot* would be treated as mode *r*, and a mode of *anything else* would be treated as mode *w*.

If the application calls *waitpid*( ) with a *pid* argument greater than zero, and it still has a *popen*( )ed stream open, it must ensure that *pid* does not refer to the process started by *popen*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

pclose( ), *pipe*( ), *sysconf*( ), *system*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *sh*

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

A statement is added to the DESCRIPTION indicating that pipe streams are byte-oriented.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The optional [EMFILE] error condition is added.

NAME

posix_fadvise — file advisory information (**ADVANCED REALTIME**)

SYNOPSIS

ADV

```
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset, size_t len, int advice);
```

DESCRIPTION

The *posix_fadvise*( ) function shall advise the implementation on the expected behavior of the application with respect to the data in the file associated with the open file descriptor, *fd*, starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the file. If *len* is zero, all data following *offset* is specified. The implementation may use this information to optimize handling of the specified data. The *posix_fadvise*( ) function shall have no effect on the semantics of other operations on the specified data, although it may affect the performance of other operations.

The advice to be applied to the data is specified by the *advice* parameter and may be one of the following values:

POSIX_FADV_NORMAL
Specifies that the application has no advice to give on its behavior with respect to the specified data. It is the default characteristic if no advice is given for an open file.

POSIX_FADV_SEQUENTIAL
Specifies that the application expects to access the specified data sequentially from lower offsets to higher offsets.

POSIX_FADV_RANDOM
Specifies that the application expects to access the specified data in a random order.

POSIX_FADV_WILLNEED
Specifies that the application expects to access the specified data in the near future.

POSIX_FADV_DONTNEED
Specifies that the application expects that it will not access the specified data in the near future.

POSIX_FADV_NOREUSE
Specifies that the application expects to access the specified data once and then not reuse it thereafter.

These values are defined in **<fcntl.h>**.

RETURN VALUE

Upon successful completion, *posix_fadvise*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_fadvise*( ) function shall fail if:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EINVAL]	The value of *advice* is invalid.
[ESPIPE]	The *fd* argument is associated with a pipe or FIFO.

EXAMPLES

None.

APPLICATION USAGE

The *posix_fadvise*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_madvise( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

posix_fallocate — file space control (**ADVANCED REALTIME**)

SYNOPSIS

ADV
```
#include <fcntl.h>

int posix_fallocate(int fd, off_t offset, size_t len);
```

DESCRIPTION

The *posix_fallocate*( ) function shall ensure that any required storage for regular file data starting at *offset* and continuing for *len* bytes is allocated on the file system storage media. If *posix_fallocate*( ) returns successfully, subsequent writes to the specified file data shall not fail due to the lack of free space on the file system storage media.

If the *offset*+*len* is beyond the current file size, then *posix_fallocate*( ) shall adjust the file size to *offset*+*len*. Otherwise, the file size shall not be changed.

It is implementation-defined whether a previous *posix_fadvise*( ) call influences allocation strategy.

Space allocated via *posix_fallocate*( ) shall be freed by a successful call to *creat*( ) or *open*( ) that truncates the size of the file. Space allocated via *posix_fallocate*( ) may be freed by a successful call to *ftruncate*( ) that reduces the file size to a size smaller than *offset*+*len*.

RETURN VALUE

Upon successful completion, *posix_fallocate*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_fallocate*( ) function shall fail if:

[EBADF]	The *fd* argument is not a valid file descriptor.
[EBADF]	The *fd* argument references a file that was opened without write permission.
[EFBIG]	The value of *offset*+*len* is greater than the maximum file size.
[EINTR]	A signal was caught during execution.
[EINVAL]	The *len* argument was zero or the *offset* argument was less than zero.
[EIO]	An I/O error occurred while reading from or writing to a file system.
[ENODEV]	The *fd* argument does not refer to a regular file.
[ENOSPC]	There is insufficient free space remaining on the file system storage media.
[ESPIPE]	The *fd* argument is associated with a pipe or FIFO.

EXAMPLES

None.

APPLICATION USAGE

The *posix_fallocate*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

creat( ), *ftruncate*( ), *open*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

posix_madvise — memory advisory information and alignment control (**ADVANCED REALTIME**)

SYNOPSIS

ADV

```
#include <sys/mman.h>

int posix_madvise(void *addr, size_t len, int advice);

```

DESCRIPTION

MF|SHM The *posix_madvise*( ) function need only be supported if either the Memory Mapped Files or the Shared Memory Objects options are supported.

The *posix_madvise*( ) function shall advise the implementation on the expected behavior of the application with respect to the data in the memory starting at address *addr*, and continuing for *len* bytes. The implementation may use this information to optimize handling of the specified data. The *posix_madvise*( ) function shall have no effect on the semantics of access to memory in the specified range, although it may affect the performance of access.

The implementation may require that *addr* be a multiple of the page size, which is the value returned by *sysconf*( ) when the name value _SC_PAGESIZE is used.

The advice to be applied to the memory range is specified by the *advice* parameter and may be one of the following values:

POSIX_MADV_NORMAL
Specifies that the application has no advice to give on its behavior with respect to the specified range. It is the default characteristic if no advice is given for a range of memory.

POSIX_MADV_SEQUENTIAL
Specifies that the application expects to access the specified range sequentially from lower addresses to higher addresses.

POSIX_MADV_RANDOM
Specifies that the application expects to access the specified range in a random order.

POSIX_MADV_WILLNEED
Specifies that the application expects to access the specified range in the near future.

POSIX_MADV_DONTNEED
Specifies that the application expects that it will not access the specified range in the near future.

These values are defined in the **<sys/mman.h>** header.

RETURN VALUE

Upon successful completion, *posix_madvise*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_madvise*( ) function shall fail if:

[EINVAL]	The value of *advice* is invalid.
[ENOMEM]	Addresses in the range starting at *addr* and continuing for *len* bytes are partly or completely outside the range allowed for the address space of the calling process.

The *posix_madvise*( ) function may fail if:

[EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf*( ) when the name value _SC_PAGESIZE is used.

[EINVAL] The value of *len* is zero.

EXAMPLES

None.

APPLICATION USAGE

The *posix_madvise*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *posix_fadvise*( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #102 is applied. |

NAME

posix_mem_offset — find offset and length of a mapped typed memory block (**ADVANCED REALTIME**)

SYNOPSIS

TYM
```
#include <sys/mman.h>

int posix_mem_offset(const void *restrict addr, size_t len,
    off_t *restrict off, size_t *restrict contig_len,
    int *restrict fildes);
```

DESCRIPTION

The *posix_mem_offset*( ) function shall return in the variable pointed to by *off* a value that identifies the offset (or location), within a memory object, of the memory block currently mapped at *addr*. The function shall return in the variable pointed to by *fildes*, the descriptor used (via *mmap*( )) to establish the mapping which contains *addr*. If that descriptor was closed since the mapping was established, the returned value of *fildes* shall be –1. The *len* argument specifies the length of the block of the memory object the user wishes the offset for; upon return, the value pointed to by *contig_len* shall equal either *len*, or the length of the largest contiguous block of the memory object that is currently mapped to the calling process starting at *addr*, whichever is smaller.

If the memory object mapped at *addr* is a typed memory object, then if the *off* and *contig_len* values obtained by calling *posix_mem_offset*( ) are used in a call to *mmap*( ) with a file descriptor that refers to the same memory pool as *fildes* (either through the same port or through a different port), and that was opened with neither the POSIX_TYPED_MEM_ALLOCATE nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag, the typed memory area that is mapped shall be exactly the same area that was mapped at *addr* in the address space of the process that called *posix_mem_offset*( ).

If the memory object specified by *fildes* is not a typed memory object, then the behavior of this function is implementation-defined.

RETURN VALUE

Upon successful completion, the *posix_mem_offset*( ) function shall return zero; otherwise, the corresponding error status value shall be returned.

ERRORS

The *posix_mem_offset*( ) function shall fail if:

[EACCES] The process has not mapped a memory object supported by this function at the given address *addr*.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mmap( ), *posix_typed_mem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

posix_memalign — aligned memory allocation (**ADVANCED REALTIME**)

SYNOPSIS

ADV

```
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment, size_t size);

```

DESCRIPTION

The *posix_memalign*( ) function shall allocate *size* bytes aligned on a boundary specified by *alignment*, and shall return a pointer to the allocated memory in *memptr*. The value of *alignment* shall be a multiple of *sizeof*(**void ***), that is also a power of two. Upon successful completion, the value pointed to by *memptr* shall be a multiple of *alignment*.

CX The *free*( ) function shall deallocate memory that has previously been allocated by *posix_memalign*( ).

RETURN VALUE

Upon successful completion, *posix_memalign*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_memalign*( ) function shall fail if:

[EINVAL]	The value of the alignment parameter is not a power of two multiple of *sizeof*(**void ***).
[ENOMEM]	There is insufficient memory available with the requested alignment.

EXAMPLES

None.

APPLICATION USAGE

The *posix_memalign*( ) function is part of the Advisory Information option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

posix_openpt — open a pseudo terminal device

SYNOPSIS

XSI

```
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

DESCRIPTION

The *posix_openpt*( ) function shall establish a connection between a master device for a pseudo-terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo-terminal.

The file status flags and file access modes of the open file description shall be set according to the value of *oflag*.

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

O_RDWR	Open for reading and writing.
O_NOCTTY	If set *posix_openpt*( ) shall not cause the terminal device to become the controlling terminal for the process.

The behavior of other values for the *oflag* argument is unspecified.

RETURN VALUE

Upon successful completion, the *posix_openpt*( ) function shall open a master pseudo-terminal device and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *posix_openpt*( ) function shall fail if:

[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
[ENFILE]	The maximum allowable number of files is currently open in the system.

The *posix_openpt*( ) function may fail if:

	[EINVAL]	The value of *oflag* is not valid.
	[EAGAIN]	Out of pseudo-terminal resources.
XSR	[ENOSR]	Out of STREAMS resources.

EXAMPLES

Opening a Pseudo-Terminal and Returning the Name of the Slave Device and a File Descriptor

```
#include <fcntl.h>
#include <stdio.h>

int masterfd, slavefd;
char *slavedevice;

masterfd = posix_openpt(O_RDWR|O_NOCTTY);

if (masterfd == -1
    || grantpt (masterfd) == -1
```

```
            || unlockpt (masterfd) == -1
            || (slavedevice = ptsname (masterfd)) == NULL)
            return -1;

        printf("slave device is: %s\n", slavedevice);

        slavefd = open(slave, O_RDWR|O_NOCTTY);
        if (slavefd < 0)
            return -1;
```

APPLICATION USAGE

This function is a method for portably obtaining a file descriptor of a master terminal device for a pseudo-terminal. The *grantpt*( ) and *ptsname*( ) functions can be used to manipulate mode and ownership permissions, and to obtain the name of the slave device, respectively.

RATIONALE

The standard developers considered the matter of adding a special device for cloning master pseudo-terminals: the **/dev/ptmx** device. However, consensus could not be reached, and it was felt that adding a new function would permit other implementations. The *posix_openpt*( ) function is designed to complement the *grantpt*( ), *ptsname*( ), and *unlockpt*( ) functions.

On implementations supporting the **/dev/ptmx** clone device, opening the master device of a pseudo-terminal is simply:

```
        mfdp = open("/dev/ptmx", oflag );
        if (mfdp < 0)
            return -1;
```

FUTURE DIRECTIONS

None.

SEE ALSO

grantpt( ), *open*( ), *ptsname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**

CHANGE HISTORY

First released in Issue 6.

NAME

posix_spawn, posix_spawnp — spawn a process (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char * const envp[restrict]);
```

DESCRIPTION

The *posix_spawn*( ) and *posix_spawnp*( ) functions shall create a new process (child process) from the specified process image. The new process image shall be constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it shall be entered as a C language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

shall be initialized as a pointer to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process image being started by the *posix_spawn*( ) or *posix_spawnp*( ) function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the child process' combined argument and environment lists is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance) whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is included in this total.

The *path* argument to *posix_spawn*( ) is a pathname that identifies the new process image file to execute.

The *file* parameter to *posix_spawnp*( ) shall be used to construct a pathname that identifies the new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be used as the pathname for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). If this environment variable is not defined, the results of the search are implementation-defined.

If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-*exec* flag FD_CLOEXEC is set (see *fcntl*( )). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see *fcntl*( )), shall remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions shall be:

1. The set of open file descriptors for the child process shall initially be the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see *fcntl*( )), shall remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process shall be changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object shall be performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set (see *fcntl*( )) shall be closed.

The **posix_spawnattr_t** spawn attributes object type is defined in **<spawn.h>**. It shall contain at least the attributes defined below. |

If the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is non-zero, then the child's process group shall be as specified in the *spawn-pgroup* attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, and the *spawn-pgroup* attribute of the same object is set to zero, then the child shall be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the *spawn-flags* attribute of the object referenced by *attrp*, the new child process shall inherit the parent's process group.

PS If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image shall initially have the scheduling policy of the calling process with the scheduling parameters specified in the *spawn-schedparam* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSCHEDULER flag is set in *spawn-flags* attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image shall initially have the scheduling policy specified in the *spawn-schedpolicy* attribute of the object referenced by *attrp* and the scheduling parameters specified in the *spawn-schedparam* attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process shall inherit the parent process' effective user ID. If this flag is set, the child process' effective user ID shall be reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process shall become that file's owner ID before the new process image begins execution. |

The POSIX_SPAWN_RESETIDS flag in the *spawn-flags* attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process shall inherit the parent process' effective group ID. If this flag is set, the child process' effective group ID shall be reset to the parent's real group ID. In either case, if the set-group-ID mode bit |

of the new process image file is set, the effective group ID of the child process shall become that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the child process shall initially have the signal mask specified in the *spawn-sigmask* attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSIGDEF flag is set in the *spawn-flags* attribute of the object referenced by *attrp*, the signals specified in the *spawn-sigdefault* attribute of the same object shall be set to their default actions in the child process. Signals set to the default action in the parent process shall be set to the default action in the child process.

Signals set to be caught by the calling process shall be set to the default action in the child process.

Except for SIGCHLD, signals set to be ignored by the calling process image shall be set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn-flags* attribute of the object referenced by *attrp* and the signals being indicated in the *spawn-sigdefault* attribute of the object referenced by *attrp*.

If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the *spawn_flags* attribute of the object referenced by *attrp* and the SIGCHLD signal being indicated in the *spawn_sigdefault* attribute of the object referenced by *attrp*.

If the value of the *attrp* pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions*, shall appear in the new process image as though *fork*( ) had been called to create a child process and then a member of the *exec* family of functions had been called by the child process to execute the new process image.

THR It is implementation-defined whether the fork handlers are run when *posix_spawn*( ) or *posix_spawnp*( ) is called.

RETURN VALUE

Upon successful completion, *posix_spawn*( ) and *posix_spawnp*( ) shall return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and shall return zero as the function return value. Otherwise, no child process shall be created, the value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number shall be returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

ERRORS

The *posix_spawn*( ) and *posix_spawnp*( ) functions may fail if:

[EINVAL]	The value specified by *file_actions* or *attrp* is invalid.

If this error occurs after the calling process successfully returns from the *posix_spawn*( ) or *posix_spawnp*( ) function, the child process may exit with exit status 127.

If *posix_spawn*( ) or *posix_spawnp*( ) fail for any of the reasons that would cause *fork*( ) or one of the *exec* family of functions to fail, an error value shall be returned as described by *fork*( ) and *exec*, respectively (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute of the object referenced by *attrp*, and *posix_spawn*( ) or *posix_spawnp*( ) fails while changing the child's process group, an error value shall be returned as described by *setpgid*( ) (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

PS If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the *spawn-flags* attribute of the object referenced by *attrp*, then if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *sched_setparam*( ) to fail, an error value shall be returned as described by *sched_setparam*( ) (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute of the object referenced by *attrp*, and if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *sched_setscheduler*( ) to fail, an error value shall be returned as described by *sched_setscheduler*( ) (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If the *file_actions* argument is not NULL, and specifies any *close*, *dup2*, or *open* actions to be performed, and if *posix_spawn*( ) or *posix_spawnp*( ) fails for any of the reasons that would cause *close*( ), *dup2*( ), or *open*( ) to fail, an error value shall be returned as described by *close*( ), *dup2*( ), and *open*( ), respectively (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127). An open file action may, by itself, result in any of the errors described by *close*( ) or *dup2*( ), in addition to those described by *open*( ).

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

The *posix_spawn*( ) function and its close relation *posix_spawnp*( ) have been introduced to overcome the following perceived difficulties with *fork*( ): the *fork*( ) function is difficult or impossible to implement without swapping or dynamic address translation. |

- Swapping is generally too slow for a realtime environment.
- Dynamic address translation is not available everywhere that POSIX might be useful. |
- Processes are too useful to simply option out of POSIX whenever it must run without address translation or other MMU services,

Thus, POSIX needs process creation and file execution primitives that can be efficiently implemented without address translation or other MMU services. |

The *posix_spawn*( ) function is implementable as a library routine, but both *posix_spawn*( ) and *posix_spawnp*( ) are designed as kernel operations. Also, although they may be an efficient replacement for many *fork*( )/*exec* pairs, their goal is to provide useful process creation primitives for systems that have difficulty with *fork*( ), not to provide drop-in replacements for *fork*( )/*exec*. |

This view of the role of *posix_spawn*( ) and *posix_spawnp*( ) influenced the design of their API. It does not attempt to provide the full functionality of *fork*( )/*exec* in which arbitrary user-specified operations of any sort are permitted between the creation of the child process and the execution of the new process image; any attempt to reach that level would need to provide a programming language as parameters. Instead, *posix_spawn*( ) and *posix_spawnp*( ) are process creation primitives like the *Start_Process* and *Start_Process_Search* Ada language bindings package *POSIX_Process_Primitives* and also like those in many operating systems that are not UNIX systems, but with some POSIX-specific additions.

To achieve its coverage goals, *posix_spawn*( ) and *posix_spawnp*( ) have control of six types of inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and whether each signal ignored in the parent will remain ignored in the child, or be reset to its default action in the child.

Control of file descriptors is required to allow an independently written child process image to access data streams opened by and even generated or read by the parent process without being specifically coded to know which parent files and file descriptors are to be used. Control of the process group ID is required to control how the child process' job control relates to that of the parent.

Control of the signal mask and signal defaulting is sufficient to support the implementation of *system*( ). Although support for *system*( ) is not explicitly one of the goals for *posix_spawn*( ) and *posix_spawnp*( ), it is covered under the ''at least 50%'' coverage goal.

The intention is that the normal file descriptor inheritance across *fork*( ), the subsequent effect of the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec* family of functions should fully specify open file inheritance. The implementation need make no decisions regarding the set of open file descriptors when the child process image begins execution, those decisions having already been made by the caller and expressed as the set of open file descriptors and their FD_CLOEXEC flags at the time of the call and the spawn file actions object specified in the call. We have been assured that in cases where the POSIX *Start_Process* Ada primitives have been implemented in a library, this method of controlling file descriptor inheritance may be implemented very easily.

We can identify several problems with *posix_spawn*( ) and *posix_spawnp*( ), but there does not appear to be a solution that introduces fewer problems. Environment modification for child process attributes not specifiable via the *attrp* or *file_actions* arguments must be done in the parent process, and since the parent generally wants to save its context, it is more costly than similar functionality with *fork*( )/*exec*. It is also complicated to modify the environment of a multi-threaded process temporarily, since all threads must agree when it is safe for the environment to be changed. However, this cost is only borne by those invocations of *posix_spawn*( ) and *posix_spawnp*( ) that use the additional functionality. Since extensive modifications are not the usual case, and are particularly unlikely in time-critical code, keeping much of the environment control out of *posix_spawn*( ) and *posix_spawnp*( ) is appropriate design.

The *posix_spawn*( ) and *posix_spawnp*( ) functions do not have all the power of *fork*( )/*exec*. This is to be expected. The *fork*( ) function is a wonderfully powerful operation. We do not expect to duplicate its functionality in a simple, fast function with no special hardware requirements. It is worth noting that *posix_spawn*( ) and *posix_spawnp*( ) are very similar to the process creation operations on many operating systems that are not UNIX systems.

Requirements

The requirements for *posix_spawn*( ) and *posix_spawnp*( ) are:

- They must be implementable without an MMU or unusual hardware.
- They must be compatible with existing POSIX standards.

Additional goals are:

- They should be efficiently implementable.
- They should be able to replace at least 50% of typical executions of *fork*( ).

- A system with *posix_spawn*( ) and *posix_spawnp*( ) and without *fork*( ) should be useful, at least for realtime applications.
- A system with *fork*( ) and the *exec* family should be able to implement *posix_spawn*( ) and *posix_spawnp*( ) as library routines.

Two-Syntax

POSIX *exec* has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the *posix_spawn**( ) functions is otherwise substantially unlike POSIX, we feel that simplicity outweighs compatibility. There are, therefore, only two names for the *posix_spawn**( ) functions.

The parameter list does not differ between *posix_spawn*( ) and *posix_spawnp*( ); *posix_spawnp*( ) interprets the second parameter more elaborately than *posix_spawn*( ).

Compatibility with POSIX.5 (Ada)

The *Start_Process* and *Start_Process_Search* procedures from the POSIX_Process_Primitives package from the Ada language binding to POSIX.1 encapsulate *fork*( ) and *exec* functionality in a manner similar to that of *posix_spawn*( ) and *posix_spawnp*( ). Originally, in keeping with our simplicity goal, the standard developers had limited the capabilities of *posix_spawn*( ) and *posix_spawnp*( ) to a subset of the capabilities of *Start_Process* and *Start_Process_Search*; certain non-default capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings working group member, the standard developers decided that *posix_spawn*( ) and *posix_spawnp*( ) should be sufficiently powerful to implement *Start_Process* and *Start_Process_Search*. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little justification for not approving the functionally-equivalent parts of a C binding. The only three capabilities provided by *posix_spawn*( ) and *posix_spawnp*( ) that are not provided by *Start_Process* and *Start_Process_Search* are optionally specifying the child's process group ID, the set of signals to be reset to default signal handling in the child process, and the child's scheduling policy and parameters.

For the Ada language binding for *Start_Process* to be implemented with *posix_spawn*( ), that binding would need to explicitly pass an empty signal mask and the parent's environment to *posix_spawn*( ) whenever the caller of *Start_Process* allowed these arguments to default, since *posix_spawn*( ) does not provide such defaults. The ability of *Start_Process* to mask user-specified signals during its execution is functionally unique to the Ada language binding and must be dealt with in the binding separately from the call to *posix_spawn*( ).

Process Group

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the *spawn-pgroup* attribute of the object referenced by *attrp*, the *setpgid*( ) mechanism will place the child process in a new process group.

Threads

Without the *posix_spawn*( ) and *posix_spawnp*( ) functions, systems without address translation can still use threads to give an abstraction of concurrency. In many cases, thread creation suffices, but it is not always a good substitute. The *posix_spawn*( ) and *posix_spawnp*( ) functions are considerably ‘‘heavier’’ than thread creation. Processes have several important attributes that threads do not. Even without address translation, a process may have base-and-bound memory protection. Each process has a process environment including security attributes and file capabilities, and powerful scheduling attributes. Processes abstract the behavior of non-uniform-memory-architecture multi-processors better than threads, and they are more convenient to use for activities that are not closely linked.

The *posix_spawn*( ) and *posix_spawnp*( ) functions may not bring support for multiple processes to every configuration. Process creation is not the only piece of operating system support required to support multiple processes. The total cost of support for multiple processes may be quite high in some circumstances. Existing practice shows that support for multiple processes is uncommon and threads are common among ‘‘tiny kernels’’. There should, therefore, probably continue to be AEPs for operating systems with only one process.

Asynchronous Error Notification

A library implementation of *posix_spawn*( ) or *posix_spawnp*( ) may not be able to detect all possible errors before it forks the child process. IEEE Std 1003.1-200x provides for an error indication returned from a child process which could not successfully complete the spawn operation via a special exit status which may be detected using the status value returned by *wait*( ) and *waitpid*( ).

The *stat_val* interface and the macros used to interpret it are not well suited to the purpose of returning API errors, but they are the only path available to a library implementation. Thus, an implementation may cause the child process to exit with exit status 127 for any error detected during the spawn process after the *posix_spawn*( ) or *posix_spawnp*( ) function has successfully returned.

The standard developers had proposed using two additional macros to interpret *stat_val*. The first, WIFSPAWNFAIL, would have detected a status that indicated that the child exited because of an error detected during the *posix_spawn*( ) or *posix_spawnp*( ) operations rather than during actual execution of the child process image; the second, WSPAWNERRNO, would have extracted the error value if WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group strongly opposed this because it would make a library implementation of *posix_spawn*( ) or *posix_spawnp*( ) dependent on kernel modifications to *waitpid*( ) to be able to embed special information in *stat_val* to indicate a spawn failure.

The 8 bits of child process exit status that are guaranteed by IEEE Std 1003.1-200x to be accessible to the waiting parent process are insufficient to disambiguate a spawn error from any other kind of error that may be returned by an arbitrary process image. No other bits of the exit status are required to be visible in *stat_val*, so these macros could not be strictly implemented at the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use of this value by *system*( ) and *popen*( ) to signal failures in these operations that occur after the function has returned but before a shell is able to execute. The exit status of 127 does not uniquely identify this class of error, nor does it provide any detailed information on the nature of the failure. Note that a kernel implementation of *posix_spawn*( ) or *posix_spawnp*( ) is permitted (and encouraged) to return any possible error as the function value, thus providing more detailed failure information to the parent process.

Thus, no special macros are available to isolate asynchronous *posix_spawn*( ) or *posix_spawnp*( ) errors. Instead, errors detected by the *posix_spawn*( ) or *posix_spawnp*( ) operations in the context of the child process before the new process image executes are reported by setting the child's exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on the *stat_val* stored by the *wait*( ) or *waitpid*( ) functions to detect spawn failures to the extent that other status values with which the child process image may exit (before the parent can conclusively determine that the child process image has begun execution) are distinct from exit status 127.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *chmod*( ), *close*( ), *dup*( ), *exec*, *exit*( ), *fcntl*( ), *fork*( ), *kill*( ), *open*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_addopen*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawn_file_actions_init*( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *sched_setparam*( ), *sched_setscheduler*( ), *setpgid*( ), *setuid*( ), *stat*( ), *times*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #103 is applied, noting that the signal default actions are changed as well as the signal mask in step 2. |

IEEE PASC Interpretation 1003.1 #132 is applied. |

NAME

posix_spawn_file_actions_addclose, posix_spawn_file_actions_addopen — add close or open action to spawn file actions object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
    file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict
    file_actions, int fildes, const char *restrict path,
    int oflag, mode_t mode);
```

DESCRIPTION

These functions shall add or delete a close or open action to a spawn file actions object.

A spawn file actions object is of type **posix_spawn_file_actions_t** (defined in **<spawn.h>**) and is used to specify a series of actions to be performed by a *posix_spawn*( ) or *posix_spawnp*( ) operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent. IEEE Std 1003.1-200x does not define comparison or assignment operators for the type **posix_spawn_file_actions_t**.

A spawn file actions object, when passed to *posix_spawn*( ) or *posix_spawnp*( ), shall specify how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation shall be as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object; additionally, when the new process image is executed, any file descriptor (from this new set) which has its FD_CLOEXEC flag set shall be closed (see *posix_spawn*( )).

The *posix_spawn_file_actions_addclose*( ) function shall add a *close* action to the object referenced by *file_actions* that shall cause the file descriptor *fildes* to be closed (as if *close*(*fildes*) had been called) when a new process is spawned using this file actions object.

The *posix_spawn_file_actions_addopen*( ) function shall add an *open* action to the object referenced by *file_actions* that shall cause the file named by *path* to be opened (as if *open*(*path*, *oflag*, *mode*) had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it shall be closed before the new file is opened.

The string described by *path* shall be copied by the *posix_spawn_file_actions_addopen*( ) function.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions shall fail if:

[EBADF]	The value specified by *fildes* is negative or greater than or equal to {OPEN_MAX}.

These functions may fail if:

[EINVAL]	The value specified by *file_actions* is invalid.
[ENOMEM]	Insufficient memory exists to add to the spawn file actions object.

It shall not be considered an error for the *fildes* argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn*( ) or *posix_spawnp*( ) operation.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

A spawn file actions object may be initialized to contain an ordered sequence of *close*( ), *dup2*( ), and *open*( ) operations to be used by *posix_spawn*( ) or *posix_spawnp*( ) to arrive at the set of open file descriptors inherited by the spawned process from the set of open file descriptors in the parent at the time of the *posix_spawn*( ) or *posix_spawnp*( ) call. It had been suggested that the *close*( ) and *dup2*( ) operations alone are sufficient to rearrange file descriptors, and that files which need to be opened for use by the spawned process can be handled either by having the calling process open them before the *posix_spawn*( ) or *posix_spawnp*( ) call (and close them after), or by passing filenames to the spawned process (in *argv*) so that it may open them itself. The standard developers recommend that applications use one of these two methods when practical, since detailed error status on a failed open operation is always available to the application this way. However, the standard developers feel that allowing a spawn file actions object to specify open operations is still appropriate because:

1. It is consistent with equivalent POSIX.5 (Ada) functionality.
2. It supports the I/O redirection paradigm commonly employed by POSIX programs designed to be invoked from a shell. When such a program is the child process, it may not be designed to open files on its own.
3. It allows file opens that might otherwise fail or violate file ownership/access rights if executed by the parent process.

Regarding 2. above, note that the spawn open file action provides to *posix_spawn*( ) and *posix_spawnp*( ) the same capability that the shell redirection operators provide to *system*( ), only without the intervening execution of a shell; for example:

```
system ("myprog <file1 3<file2");
```

Regarding 3. above, note that if the calling process needs to open one or more files for access by the spawned process, but has insufficient spare file descriptors, then the open action is necessary to allow the *open*( ) to occur in the context of the child process after other file descriptors have been closed (that must remain open in the parent).

Additionally, if a parent is executed from a file having a ''set-user-id'' mode bit set and the POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent process will (possibly incorrectly) have the parent's effective user ID as its owner, whereas a file created via an *open*( ) action during *posix_spawn*( ) or *posix_spawnp*( ) will have the parent's real ID as its owner; and an open by the parent process may successfully open a file to which the real user should not have access or fail to open a file to which the real user should have access.

File Descriptor Mapping

The standard developers had originally proposed using an array which specified the mapping of child file descriptors back to those of the parent. It was pointed out by the ballot group that it is not possible to reshuffle file descriptors arbitrarily in a library implementation of *posix_spawn*( ) or *posix_spawnp*( ) without provision for one or more spare file descriptor entries (which simply may not be available). Such an array requires that an implementation develop a complex strategy to achieve the desired mapping without inadvertently closing the wrong file descriptor at the wrong time.

It was noted by a member of the Ada Language Bindings working group that the approved Ada Language *Start_Process* family of POSIX process primitives use a caller-specified set of file actions to alter the normal *fork*( )/*exec* semantics for inheritance of file descriptors in a very flexible way, yet no such problems exist because the burden of determining how to achieve the final file descriptor mapping is completely on the application. Furthermore, although the file actions interface appears frightening at first glance, it is actually quite simple to implement in either a library or the kernel.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *dup*( ), *open*( ), *posix_spawn*( ), *posix_spawn_file_actions_adddup2*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #105 is applied, adding a note to the DESCRIPTION that the string pointed to by *path* is copied by the *posix_spawn_file_actions_addopen*( ) function.

NAME

posix_spawn_file_actions_adddup2 — add dup2 action to spawn file actions object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
    file_actions, int fildes, int newfildes);
```

DESCRIPTION

The *posix_spawn_file_actions_adddup2*( ) function shall add a *dup2*( ) action to the object referenced by *file_actions* that shall cause the file descriptor *fildes* to be duplicated as *newfildes* (as if *dup2*(*fildes*, *newfildes*) had been called) when a new process is spawned using this file actions object.

A spawn file actions object is as defined in *posix_spawn_file_actions_addclose*( ).

RETURN VALUE

Upon successful completion, the *posix_spawn_file_actions_adddup2*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_spawn_file_actions_adddup2*( ) function shall fail if:

[EBADF]	The value specified by *fildes* or *newfildes* is negative or greater than or equal to {OPEN_MAX}.
[ENOMEM]	Insufficient memory exists to add to the spawn file actions object.

The *posix_spawn_file_actions_adddup2*( ) function may fail if:

[EINVAL]	The value specified by *file_actions* is invalid.

It shall not be considered an error for the *fildes* argument passed to the *posix_spawn_file_actions_adddup2*( ) function to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn*( ) or *posix_spawnp*( ) operation.

EXAMPLES

None.

APPLICATION USAGE

The *posix_spawn_file_actions_adddup2*( ) function is part of the Spawn option and need not be provided on all implementations.

RATIONALE

Refer to the RATIONALE in *posix_spawn_file_actions_addclose*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

dup( ), *posix_spawn*( ), *posix_spawn_file_actions_addclose*( ), *posix_spawn_file_actions_destroy*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #104 is applied, noting that the [EBADF] error can apply to the *newfildes* argument in addition to *fildes*.

NAME

posix_spawn_file_actions_addopen — add open action to spawn file actions object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *restrict
    file_actions, int fildes, const char *restrict path,
    int oflag, mode_t mode);

```

DESCRIPTION

Refer to *posix_spawn_file_actions_addclose*( ).

NAME

posix_spawn_file_actions_destroy, posix_spawn_file_actions_init — destroy and initialize spawn file actions object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
    file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);
```

DESCRIPTION

The *posix_spawn_file_actions_destroy*( ) function shall destroy the object referenced by *file_actions*; the object becomes, in effect, uninitialized. An implementation may cause *posix_spawn_file_actions_destroy*( ) to set the object referenced by *file_actions* to an invalid value. A destroyed spawn file actions object can be reinitialized using *posix_spawn_file_actions_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

The *posix_spawn_file_actions_init*( ) function shall initialize the object referenced by *file_actions* to contain no file actions for *posix_spawn*( ) or *posix_spawnp*( ) to perform.

A spawn file actions object is as defined in *posix_spawn_file_actions_addclose*( ).

The effect of initializing an already initialized spawn file actions object is undefined.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_spawn_file_actions_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn file actions object.

The *posix_spawn_file_actions_destroy*( ) function may fail if:

[EINVAL] The value specified by *file_actions* is invalid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

Refer to the RATIONALE in *posix_spawn_file_actions_addclose*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

posix_spawn_file_actions_init — initialize spawn file actions object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
    file_actions);
```

DESCRIPTION

Refer to *posix_spawn_file_actions_destroy*( ).

NAME

posix_spawnattr_destroy, posix_spawnattr_init — destroy and initialize spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

DESCRIPTION

The *posix_spawnattr_destroy*( ) function shall destroy a spawn attributes object. A destroyed *attr* attributes object can be reinitialized using *posix_spawnattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause *posix_spawnattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

The *posix_spawnattr_init*( ) function shall initialize a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. Results are undefined if *posix_spawnattr_init*( ) is called specifying an already initialized *attr* attributes object.

A spawn attributes object is of type **posix_spawnattr_t** (defined in **<spawn.h>**) and is used to specify the inheritance of process attributes across a spawn operation. IEEE Std 1003.1-200x does not define comparison or assignment operators for the type **posix_spawnattr_t**.

Each implementation shall document the individual attributes it uses and their default values unless these values are defined by IEEE Std 1003.1-200x. Attributes not defined by IEEE Std 1003.1-200x, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of *posix_spawn*( ) or *posix_spawnp*( ). After a spawn attributes object has been used to spawn a process by a call to a *posix_spawn*( ) or *posix_spawnp*( ), any function affecting the attributes object (including destruction) shall not affect any process that has been spawned in this way.

RETURN VALUE

Upon successful completion, *posix_spawnattr_destroy*( ) and *posix_spawnattr_init*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *posix_spawnattr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

The *posix_spawnattr_destroy*( ) function may fail if:

[EINVAL] The value specified by attr is invalid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

The original spawn interface proposed in IEEE Std 1003.1-200x defined the attributes that specify the inheritance of process attributes across a spawn operation as a structure. In order to be able to separate optional individual attributes under their appropriate options (that is, the *spawn-schedparam* and *spawn-schedpolicy* attributes depending upon the Process Scheduling option), and also for extensibility and consistency with the newer POSIX interfaces, the attributes interface has been changed to an opaque data type. This interface now consists of the type **posix_spawnattr_t**, representing a spawn attributes object, together with associated functions to initialize or destroy the attributes object, and to set or get each individual attribute. Although the new object-oriented interface is more verbose than the original structure, it is simple to use, more extensible, and easy to implement.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #106 is applied, noting that the effect of initializing an already initialized spawn attributes option is undefined. |

NAME

posix_spawnattr_getflags, posix_spawnattr_setflags — get and set spawn-flags attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN
```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
    short *restrict flags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

DESCRIPTION

The *posix_spawnattr_getflags*( ) function shall obtain the value of the *spawn-flags* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setflags*( ) function shall set the *spawn-flags* attribute in an initialized attributes object referenced by *attr*.

The *spawn-flags* attribute is used to indicate which process attributes are to be changed in the new process image when invoking *posix_spawn*( ) or *posix_spawnp*( ). It is the bitwise-inclusive OR of zero or more of the following flags:

POSIX_SPAWN_RESETIDS
POSIX_SPAWN_SETPGROUP
POSIX_SPAWN_SETSIGDEF
POSIX_SPAWN_SETSIGMASK
PS POSIX_SPAWN_SETSCHEDPARAM
POSIX_SPAWN_SETSCHEDULER

These flags are defined in **<spawn.h>**. The default value of this attribute shall be as if no flags were set.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getflags*( ) shall return zero and store the value of the *spawn-flags* attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setflags*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *posix_spawnattr_setflags*( ) function may fail if:

[EINVAL]	The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getpgroup, posix_spawnattr_setpgroup — get and set spawn-pgroup attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
    pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

DESCRIPTION

The *posix_spawnattr_getpgroup*( ) function shall obtain the value of the *spawn-pgroup* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setpgroup*( ) function shall set the *spawn-pgroup* attribute in an initialized attributes object referenced by *attr*.

The *spawn-pgroup* attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the *spawn-flags* attribute). The default value of this attribute shall be zero.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getpgroup*( ) shall return zero and store the value of the *spawn-pgroup* attribute of *attr* into the object referenced by the *pgroup* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setpgroup*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *posix_spawnattr_setpgroup*( ) function may fail if:

[EINVAL] The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getschedparam, posix_spawnattr_setschedparam — get and set spawn-schedparam attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN PS

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(
    const posix_spawnattr_t *restrict attr,
    struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```

DESCRIPTION

The *posix_spawnattr_getschedparam*( ) function shall obtain the value of the *spawn-schedparam* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setschedparam*( ) function shall set the *spawn-schedparam* attribute in an initialized attributes object referenced by *attr*.

The *spawn-schedparam* attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or POSIX_SPAWN_SETSCHEDPARAM is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getschedparam*( ) shall return zero and store the value of the *spawn-schedparam* attribute of *attr* into the object referenced by the *schedparam* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setschedparam*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *posix_spawnattr_setschedparam*( ) function may fail if:

[EINVAL] The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn and Process Scheduling options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getschedpolicy, posix_spawnattr_setschedpolicy — get and set spawn-schedpolicy attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN PS

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(
    const posix_spawnattr_t *restrict attr,
    int *restrict schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```

DESCRIPTION

The *posix_spawnattr_getschedpolicy*( ) function shall obtain the value of the *spawn-schedpolicy* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setschedpolicy*( ) function shall set the *spawn-schedpolicy* attribute in an initialized attributes object referenced by *attr*.

The *spawn-schedpolicy* attribute represents the scheduling policy to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getschedpolicy*( ) shall return zero and store the value of the *spawn-schedpolicy* attribute of *attr* into the object referenced by the *schedpolicy* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setschedpolicy*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *posix_spawnattr_setschedpolicy*( ) function may fail if:

[EINVAL]	The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn and Process Scheduling options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getsigdefault, posix_spawnattr_setsigdefault — get and set spawn-sigdefault attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(
    const posix_spawnattr_t *restrict attr,
    sigset_t *restrict sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```

DESCRIPTION

The *posix_spawnattr_getsigdefault*( ) function shall obtain the value of the *spawn-sigdefault* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setsigdefault*( ) function shall set the *spawn-sigdefault* attribute in an initialized attributes object referenced by *attr*.

The *spawn-sigdefault* attribute represents the set of signals to be forced to default signal handling in the new process image (if POSIX_SPAWN_SETSIGDEF is set in the *spawn-flags* attribute) by a spawn operation. The default value of this attribute shall be an empty signal set.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getsigdefault*( ) shall return zero and store the value of the *spawn-sigdefault* attribute of *attr* into the object referenced by the *sigdefault* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setsigdefault*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *posix_spawnattr_setsigdefault*( ) function may fail if:

[EINVAL]	The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_getsigmask*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnattr_setsigmask*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_getsigmask, posix_spawnattr_setsigmask — get and set spawn-sigmask attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
    sigset_t *restrict sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```

DESCRIPTION

The *posix_spawnattr_getsigmask*( ) function shall obtain the value of the *spawn-sigmask* attribute from the attributes object referenced by *attr*.

The *posix_spawnattr_setsigmask*( ) function shall set the *spawn-sigmask* attribute in an initialized attributes object referenced by *attr*.

The *spawn-sigmask* attribute represents the signal mask in effect in the new process image of a spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the *spawn-flags* attribute). The default value of this attribute is unspecified.

RETURN VALUE

Upon successful completion, *posix_spawnattr_getsigmask*( ) shall return zero and store the value of the *spawn-sigmask* attribute of *attr* into the object referenced by the *sigmask* parameter; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *posix_spawnattr_setsigmask*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *posix_spawnattr_setsigmask*( ) function may fail if:

[EINVAL]	The value of the attribute being set is not valid.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_spawn( ), *posix_spawnattr_destroy*( ), *posix_spawnattr_init*( ), *posix_spawnattr_getsigdefault*( ), *posix_spawnattr_getflags*( ), *posix_spawnattr_getpgroup*( ), *posix_spawnattr_getschedparam*( ), *posix_spawnattr_getschedpolicy*( ), *posix_spawnattr_setsigdefault*( ), *posix_spawnattr_setflags*( ), *posix_spawnattr_setpgroup*( ), *posix_spawnattr_setschedparam*( ), *posix_spawnattr_setschedpolicy*( ), *posix_spawnp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<spawn.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

posix_spawnattr_init — initialize spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN
```
#include <spawn.h>

int posix_spawnattr_init(posix_spawnattr_t *attr);

```

DESCRIPTION

Refer to *posix_spawnattr_destroy*( ).

NAME

posix_spawnattr_setflags — set spawn-flags attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

DESCRIPTION

Refer to *posix_spawnattr_getflags*( ).

NAME

posix_spawnattr_setpgroup — set spawn-pgroup attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

DESCRIPTION

Refer to *posix_spawnattr_getpgroup*( ).

NAME

posix_spawnattr_setschedparam — set spawn-schedparam attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN PS

```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
    const struct sched_param *restrict schedparam);
```

DESCRIPTION

Refer to *posix_spawnattr_getschedparam*( ).

NAME

posix_spawnattr_setschedpolicy — set spawn-schedpolicy attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN PS

```
#include <sched.h>
#include <spawn.h>

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
    int schedpolicy);
```

DESCRIPTION

Refer to *posix_spawnattr_getschedpolicy*( ).

NAME

posix_spawnattr_setsigdefault — set spawn-sigdefault attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN
```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigdefault);
```

DESCRIPTION

Refer to *posix_spawnattr_getsigdefault*( ).

NAME

posix_spawnattr_setsigmask — set spawn-sigmask attribute of spawn attributes object (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
    const sigset_t *restrict sigmask);
```

DESCRIPTION

Refer to *posix_spawnattr_getsigmask*( ).

NAME

posix_spawnp — spawn a process (**ADVANCED REALTIME**)

SYNOPSIS

SPN

```
#include <spawn.h>

int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
```

DESCRIPTION

Refer to *posix_spawn*( ).

NAME

posix_trace_attr_destroy, posix_trace_attr_init — trace stream attributes object destroy and initialization (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_attr_destroy(trace_attr_t *attr);
int posix_trace_attr_init(trace_attr_t *attr);
```

DESCRIPTION

The *posix_trace_attr_destroy*( ) function shall destroy an initialized trace attributes object. A destroyed *attr* attributes object can be reinitialized using *posix_trace_attr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

The *posix_trace_attr_init*( ) function shall initialize a trace attributes object *attr* with the default value for all of the individual attributes used by a given implementation. The read-only *generation-version* and *clock-resolution* attributes of the newly initialized trace attributes object shall be set to their appropriate values (see Section 2.11.1.2 (on page 525)).

Results are undefined if *posix_trace_attr_init*( ) is called specifying an already initialized *attr* attributes object.

Implementations may add extensions to the trace attributes object structure as permitted in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

The resulting attributes object (possibly modified by setting individual attributes values), when used by *posix_trace_create*( ), defines the attributes of the trace stream created. A single attributes object can be used in multiple calls to *posix_trace_create*( ). After one or more trace streams have been created using an attributes object, any function affecting that attributes object, including destruction, shall not affect any trace stream previously created. An initialized attributes object also serves to receive the attributes of an existing trace stream or trace log when calling the *posix_trace_get_attr*( ) function.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

ERRORS

The *posix_trace_attr_destroy*( ) function may fail if:

[EINVAL] The value of *attr* is invalid.

The *posix_trace_attr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the trace attributes object.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_create( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

IEEE PASC Interpretation 1003.1 #123 is applied. |

NAME

posix_trace_attr_getclockres, posix_trace_attr_getcreatetime, posix_trace_attr_getgenversion, posix_trace_attr_getname, posix_trace_attr_setname — retrieve and set information about a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getclockres(const trace_attr_t *attr,
    struct timespec *resolution);
int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
    struct timespec *createtime);

#include <trace.h>

int posix_trace_attr_getgenversion(const trace_attr_t *attr,
    char *genversion);
int posix_trace_attr_getname(const trace_attr_t *attr,
    char *tracename);
int posix_trace_attr_setname(trace_attr_t *attr,
    const char *tracename);
```

DESCRIPTION

The *posix_trace_attr_getclockres*( ) function shall copy the clock resolution of the clock used to generate timestamps from the *clock-resolution* attribute of the attributes object pointed to by the *attr* argument into the structure pointed to by the *resolution* argument.

The *posix_trace_attr_getcreatetime*( ) function shall copy the trace stream creation time from the *creation-time* attribute of the attributes object pointed to by the *attr* argument into the structure pointed to by the *createtime* argument. The *creation-time* attribute shall represent the time of creation of the trace stream.

The *posix_trace_attr_getgenversion*( ) function shall copy the string containing version information from the *generation-version* attribute of the attributes object pointed to by the *attr* argument into the string pointed to by the *genversion* argument. The *genversion* argument shall be the address of a character array which can store at least {TRACE_NAME_MAX} characters.

The *posix_trace_attr_getname*( ) function shall copy the string containing the trace name from the *trace-name* attribute of the attributes object pointed to by the *attr* argument into the string pointed to by the *tracename* argument. The *tracename* argument shall be the address of a character array which can store at least {TRACE_NAME_MAX} characters.

The *posix_trace_attr_setname*( ) function shall set the name in the *trace-name* attribute of the attributes object pointed to by the *attr* argument, using the trace name string supplied by the *tracename* argument. If the supplied string contains more than {TRACE_NAME_MAX} characters, the name copied into the *trace-name* attribute may be truncated to one less than the length of {TRACE_NAME_MAX} characters. The default value is a null string.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, the *posix_trace_attr_getclockres*( ) function stores the *clock-resolution* attribute value in the object pointed to by *resolution*. Otherwise, the content of this object is unspecified.

If successful, the *posix_trace_attr_getcreatetime*( ) function stores the trace stream creation time in the object pointed to by *createtime*. Otherwise, the content of this object is unspecified.

If successful, the *posix_trace_attr_getgenversion*( ) function stores the trace version information in the string pointed to by *genversion*. Otherwise, the content of this string is unspecified.

If successful, the *posix_trace_attr_getname*( ) function stores the trace name in the string pointed to by *tracename*. Otherwise, the content of this string is unspecified.

ERRORS

The *posix_trace_attr_getclockres*( ), *posix_trace_attr_getcreatetime*( ), *posix_trace_attr_getgenversion*( ), and *posix_trace_attr_getname*( ) functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_get_attr*( ), *uname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

NAME

posix_trace_attr_getcreatetime — retrieve and set information about a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getcreatetime(const trace_attr_t *attr,
    struct timespec *createtime);
```

DESCRIPTION

Refer to *posix_trace_attr_getclockres*( ).

NAME

posix_trace_attr_getgenversion — retrieve and set information about a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_attr_getgenversion(const trace_attr_t *attr,
    char *genversion);
```

DESCRIPTION

Refer to *posix_trace_attr_getclockres*( ).

NAME

posix_trace_attr_getinherited, posix_trace_attr_getlogfullpolicy, posix_trace_attr_getstreamfullpolicy, posix_trace_attr_setinherited, posix_trace_attr_setlogfullpolicy, posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>
```

TRC TRI
```
int posix_trace_attr_getinherited(const trace_attr_t *restrict attr,
    int *restrict inheritancepolicy);
```
TRC TRL
```
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
    int *restrict logpolicy);
```
TRC
```
int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
    int *streampolicy);
```
TRC TRI
```
int posix_trace_attr_setinherited(trace_attr_t *attr,
    int inheritancepolicy);
```
TRC TRL
```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```
TRC
```
int posix_trace_attr_setstreamfullpolicy(trace_attr_t *attr,
    int streampolicy);
```

DESCRIPTION

TRI The *posix_trace_attr_getinherited*( ) and *posix_trace_attr_setinherited*( ) functions, respectively, shall get and set the inheritance policy stored in the *inheritance* attribute for traced processes across the *fork*( ) and *spawn*( ) operations. The *inheritance* attribute of the attributes object pointed to by the *attr* argument shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_CLOSE_FOR_CHILD
: After a *fork*( ) or *spawn*( ) operation, the child shall not be traced, and tracing of the parent shall continue.

POSIX_TRACE_INHERITED
: After a *fork*( ) or *spawn*( ) operation, if the parent is being traced, its child shall be concurrently traced using the same trace stream.

The default value for the *inheritance* attribute is POSIX_TRACE_CLOSE_FOR_CHILD.

TRL The *posix_trace_attr_getlogfullpolicy*( ) and *posix_trace_attr_setlogfullpolicy*( ) functions, respectively, shall get and set the trace log full policy stored in the *log-full-policy* attribute of the attributes object pointed to by the *attr* argument.

The *log-full-policy* attribute shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_LOOP
: The trace log shall loop until the associated trace stream is stopped. This policy means that when the trace log gets full, the file system shall reuse the resources allocated to the oldest trace events that were recorded. In this way, the trace log will always contain the most recent trace events flushed.

POSIX_TRACE_UNTIL_FULL
: The trace stream shall be flushed to the trace log until the trace log is full. This condition can be deduced from the *posix_log_full_status* member status (see the *posix_trace_status_info*( ) function). The last recorded trace event shall be the POSIX_TRACE_STOP trace event.

POSIX_TRACE_APPEND

The associated trace stream shall be flushed to the trace log without log size limitation. If the application specifies POSIX_TRACE_APPEND, the implementation shall ignore the *log-max-size* attribute.

The default value for the *log-full-policy* attribute is POSIX_TRACE_LOOP.

The *posix_trace_attr_getstreamfullpolicy*( ) and *posix_trace_attr_setstreamfullpolicy*( ) functions, respectively, shall get and set the trace stream full policy stored in the *stream-full-policy* attribute of the attributes object pointed to by the *attr* argument.

The *stream-full-policy* attribute shall be set to one of the following values defined by manifest constants in the **<trace.h>** header:

POSIX_TRACE_LOOP

The trace stream shall loop until explicitly stopped by the *posix_trace_stop*( ) function. This policy means that when the trace stream is full, the trace system shall reuse the resources allocated to the oldest trace events recorded. In this way, the trace stream will always contain the most recent trace events recorded.

POSIX_TRACE_UNTIL_FULL

The trace stream will run until the trace stream resources are exhausted. Then the trace stream will stop. This condition can be deduced from *posix_stream_status* and *posix_stream_full_status* statuses (see the *posix_trace_status_info*( ) function). When this trace stream is read, a POSIX_TRACE_STOP trace event shall be reported after reporting the last recorded trace event. The trace system shall reuse the resources allocated to any trace events already reported—see the *posix_trace_getnext_event*( ), *posix_trace_trygetnext_event*( ), and *posix_trace_timedgetnext_event*( ) functions—or already flushed for an active trace stream with log if the Trace Log option is supported; see the *posix_trace_flush*( ) function. The trace system shall restart the trace stream when it is empty and may restart it sooner. A POSIX_TRACE_START trace event shall be reported before reporting the next recorded trace event.

TRL POSIX_TRACE_FLUSH

If the Trace Log option is supported, this policy is identical to the POSIX_TRACE_UNTIL_FULL trace stream full policy except that the trace stream shall be flushed regularly as if *posix_trace_flush*( ) had been explicitly called. Defining this policy for an active trace stream without log shall be invalid.

The default value for the *stream-full-policy* attribute shall be POSIX_TRACE_LOOP for an active trace stream without log.

TRL If the Trace Log option is supported, the default value for the *stream-full-policy* attribute shall be POSIX_TRACE_FLUSH for an active trace stream with log.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRI If successful, the *posix_trace_attr_getinherited*( ) function shall store the *inheritance* attribute value in the object pointed to by *inheritancepolicy*. Otherwise, the content of this object is undefined.

TRL If successful, the *posix_trace_attr_getlogfullpolicy*( ) function shall store the *log-full-policy* attribute value in the object pointed to by *logpolicy*. Otherwise, the content of this object is undefined.

If successful, the *posix_trace_attr_getstreamfullpolicy*( ) function shall store the *stream-full-policy* attribute value in the object pointed to by *streampolicy*. Otherwise, the content of this object is undefined.

ERRORS

These functions may fail if:

[EINVAL] The value specified by at least one of the arguments is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), *posix_trace_getnext_event*( ), *posix_trace_start*( ), **posix_trace_status_info Structure**, *posix_trace_timedgetnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

NAME

posix_trace_attr_getlogfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>
```

TRC TRL

```
int posix_trace_attr_getlogfullpolicy(const trace_attr_t *restrict attr,
    int *restrict logpolicy);
```

DESCRIPTION

Refer to *posix_trace_attr_getinherited*( ).

NAME

posix_trace_attr_getlogsize, posix_trace_attr_getmaxdatasize, posix_trace_attr_getmaxsystemeventsize, posix_trace_attr_getmaxusereventsize, posix_trace_attr_getstreamsize, posix_trace_attr_setlogsize, posix_trace_attr_setmaxdatasize, posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getlogsize(const trace_attr_t *restrict attr,
    size_t *restrict logsize);
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
    size_t *restrict maxdatasize);
int posix_trace_attr_getmaxsystemeventsize(
    const trace_attr_t *restrict attr,
    size_t *restrict eventsize);
int posix_trace_attr_getmaxusereventsize(
    const trace_attr_t *restrict attr,
    size_t data_len, size_t *restrict eventsize);
int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
int posix_trace_attr_setlogsize(trace_attr_t *attr,
    size_t logsize);
int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
    size_t maxdatasize);
int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);
```

DESCRIPTION

The *posix_trace_attr_getlogsize*( ) function shall copy the log size, in bytes, from the *log-max-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *logsize* argument. This log size is the maximum total of bytes that shall be allocated for system and user trace events in the trace log. The default value for the *log-max-size* attribute is implementation-defined.

The *posix_trace_attr_setlogsize*( ) function shall set the maximum allowed size, in bytes, in the *log-max-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *logsize* argument.

The trace log size shall be used if the *log-full-policy* attribute is set to POSIX_TRACE_LOOP or POSIX_TRACE_UNTIL_FULL. If the *log-full-policy* attribute is set to POSIX_TRACE_APPEND, the implementation shall ignore the *log-max-size* attribute.

The *posix_trace_attr_getmaxdatasize*( ) function shall copy the maximum user trace event data size, in bytes, from the *max-data-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *maxdatasize* argument. The default value for the *max-data-size* attribute is implementation-defined.

The *posix_trace_attr_getmaxsystemeventsize*( ) function shall calculate the maximum memory size, in bytes, required to store a single system trace event. This value is calculated for the trace stream attributes object pointed to by the *attr* argument and is returned in the variable pointed to by the *eventsize* argument.

The values returned as the maximum memory sizes of the user and system trace events shall be such that if the sum of the maximum memory sizes of a set of the trace events that may be recorded in a trace stream is less than or equal to the *stream-min-size* attribute of that trace stream, the system provides the necessary resources for recording all those trace events, without loss.

The *posix_trace_attr_getmaxusereventsize*( ) function shall calculate the maximum memory size, in bytes, required to store a single user trace event generated by a call to *posix_trace_event*( ) with a *data_len* parameter equal to the *data_len* value specified in this call. This value is calculated for the trace stream attributes object pointed to by the *attr* argument and is returned in the variable pointed to by the *eventsize* argument.

The *posix_trace_attr_getstreamsize*( ) function shall copy the stream size, in bytes, from the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument into the variable pointed to by the *streamsize* argument.

This stream size is the current total memory size reserved for system and user trace events in the trace stream. The default value for the *stream-min-size* attribute is implementation-defined. The stream size refers to memory used to store trace event records. Other stream data (for example, trace attribute values) shall not be included in this size.

The *posix_trace_attr_setmaxdatasize*( ) function shall set the maximum allowed size, in bytes, in the *max-data-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *maxdatasize* argument. This maximum size is the maximum allowed size for the user data argument which may be passed to *posix_trace_event*( ). The implementation shall be allowed to truncate data passed to *trace_user_event* which is longer than *maxdatasize*.

The *posix_trace_attr_setstreamsize*( ) function shall set the minimum allowed size, in bytes, in the *stream-min-size* attribute of the attributes object pointed to by the *attr* argument, using the size value supplied by the *streamsize* argument.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_attr_getlogsize*( ) function stores the maximum trace log allowed size in the object pointed to by *logsize*, if successful.

The *posix_trace_attr_getmaxdatasize*( ) function stores the maximum trace event record memory size in the object pointed to by *maxdatasize*, if successful.

The *posix_trace_attr_getmaxsystemeventsize*( ) function stores the maximum memory size to store a single system trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getmaxusereventsize*( ) function stores the maximum memory size to store a single user trace event in the object pointed to by *eventsize*, if successful.

The *posix_trace_attr_getstreamsize*( ) function stores the maximum trace stream allowed size in the object pointed to by *streamsize*, if successful.

ERRORS

These functions may fail if:

[EINVAL] The value specified by one of the arguments is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_event*( ), *posix_trace_get_attr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE Std 1003.1q-2000.

NAME

posix_trace_attr_getmaxdatasize, posix_trace_attr_getmaxsystemeventsize, posix_trace_attr_getmaxusereventsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

TRC

```
#include <sys/types.h>
#include <trace.h>
```

TRC

```
int posix_trace_attr_getmaxdatasize(const trace_attr_t *restrict attr,
    size_t *restrict maxdatasize);
int posix_trace_attr_getmaxsystemeventsize(
    const trace_attr_t *restrict attr,
    size_t *restrict eventsize);
int posix_trace_attr_getmaxusereventsize(
    const trace_attr_t *restrict attr,
    size_t data_len, size_t *restrict eventsize);
```

DESCRIPTION

Refer to *posix_trace_attr_getlogsize*( ).

NAME

posix_trace_attr_getname — retrieve and set information about a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_attr_getname(const trace_attr_t *attr,
    char *tracename);
```

DESCRIPTION

Refer to *posix_trace_attr_getclockres*( ).

NAME

posix_trace_attr_getstreamfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>

int posix_trace_attr_getstreamfullpolicy(const trace_attr_t *attr,
    int *streampolicy);
```

DESCRIPTION

Refer to *posix_trace_attr_getinherited*( ).

NAME

posix_trace_attr_getstreamsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

TRC

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getstreamsize(const trace_attr_t *restrict attr,
    size_t *restrict streamsize);
```

DESCRIPTION

Refer to *posix_trace_attr_getlogsize*( ).

NAME

posix_trace_attr_init — trace stream attributes object initialization (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_attr_init(trace_attr_t *attr);
```

DESCRIPTION

Refer to *posix_trace_attr_destroy*( ).

NAME

posix_trace_attr_setinherited — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>
```

TRC TRI

```
int posix_trace_attr_setinherited(trace_attr_t *attr,
    int inheritancepolicy);
```

DESCRIPTION

Refer to *posix_trace_attr_getinherited*( ).

NAME

posix_trace_attr_setlogfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>
```

TRC TRL
```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```

DESCRIPTION

Refer to *posix_trace_attr_getinherited*( ).

NAME

posix_trace_attr_setlogsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

TRC

```
#include <sys/types.h>
#include <trace.h>
```

TRC TRL

```
int posix_trace_attr_setlogsize(trace_attr_t *attr,
    size_t logsize);
```

DESCRIPTION

Refer to *posix_trace_attr_getlogsize*( ).

NAME

posix_trace_attr_setmaxdatasize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setmaxdatasize(trace_attr_t *attr,
    size_t maxdatasize);
```

DESCRIPTION

Refer to *posix_trace_attr_getlogsize*( ).

NAME

posix_trace_attr_setname — retrieve and set information about a trace stream (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>

int posix_trace_attr_setname(trace_attr_t *attr,
    const char *tracename);
```

DESCRIPTION

Refer to *posix_trace_attr_getclockres*( ).

NAME

posix_trace_attr_setstreamfullpolicy — retrieve and set the behavior of a trace stream (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>
```

TRC TRL

```
int posix_trace_attr_setlogfullpolicy(trace_attr_t *attr,
    int logpolicy);
```

DESCRIPTION

Refer to *posix_trace_attr_getinherited*( ).

NAME

posix_trace_attr_setstreamsize — retrieve and set trace stream size attributes (**TRACING**)

SYNOPSIS

TRC

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setstreamsize(trace_attr_t *attr,
    size_t streamsize);
```

DESCRIPTION

Refer to *posix_trace_attr_getlogsize*( ).

NAME

posix_trace_clear — clear trace stream and trace log (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_clear(trace_id_t trid);

```

DESCRIPTION

The *posix_trace_clear*( ) function shall reinitialize the trace stream identified by the argument *trid* as if it were returning from the *posix_trace_create*( ) function, except that the same allocated resources shall be reused, the mapping of trace event type identifiers to trace event names shall be unchanged, and the trace stream status shall remain unchanged (that is, if it was running, it remains running and if it was suspended, it remains suspended).

All trace events in the trace stream recorded before the call to *posix_trace_clear*( ) shall be lost. The *posix_stream_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that all trace events that occurred during the *posix_trace_clear*( ) call are recorded; the behavior with respect to trace points that may occur during this call, is unspecified.

TRL If the Trace Log option is supported and the trace stream has been created with a log, the *posix_trace_clear*( ) function shall reinitialize the trace stream with the same behavior as if the trace stream was created without the log, plus it shall reinitialize the trace log associated with the trace stream identified by the argument *trid* as if it were returning from the *posix_trace_create_withlog*( ) function, except that the same allocated resources, for the trace log, may be reused and the associated trace stream status remains unchanged. The first trace event recorded in the trace log after the call to *posix_trace_clear*( ) shall be the same as the first trace event recorded in the active trace stream after the call to *posix_trace_clear*( ). The *posix_log_full_status* status shall be set to POSIX_TRACE_NOT_FULL. There is no guarantee that all trace events that occurred during the *posix_trace_clear*( ) call are recorded in the trace log; the behavior with respect to trace points that may occur during this call is unspecified. If the log full policy is POSIX_TRACE_APPEND, the effect of a call to this function is unspecified for the trace log associated with the trace stream identified by the *trid* argument.

RETURN VALUE

Upon successful completion, the *posix_trace_clear*( ) function shall return a value of zero. Otherwise, it shall return the corresponding error number.

ERRORS

The *posix_trace_clear*( ) function shall fail if:

[EINVAL]	The value of the *trid* argument does not correspond to an active trace stream.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_attr_init( ), *posix_trace_create*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

NAME

posix_trace_close, posix_trace_open, posix_trace_rewind — trace log management (**TRACING**)

SYNOPSIS

TRC TRL

```
#include <trace.h>

int posix_trace_close(trace_id_t trid);
int posix_trace_open(int file_desc, trace_id_t *trid);
int posix_trace_rewind(trace_id_t trid);

```

DESCRIPTION

The *posix_trace_close*( ) function shall deallocate the trace log identifier indicated by *trid*, and all of its associated resources. If there is no valid trace log pointed to by the *trid*, this function shall fail.

The *posix_trace_open*( ) function shall allocate the necessary resources and establishes the connection between a trace log identified by the *file_desc* argument and a trace stream identifier identified by the object pointed to by the *trid* argument. The *file_desc* argument should be a valid open file descriptor that corresponds to a trace log. The *file_desc* argument shall be open for reading. The current trace event timestamp, which specifies the timestamp of the trace event that will be read by the next call to *posix_trace_getnext_event*( ), shall be set to the timestamp of the oldest trace event recorded in the trace log identified by *trid*.

The *posix_trace_open*( ) function shall return a trace stream identifier in the variable pointed to by the *trid* argument, that may only be used by the following functions:

posix_trace_close( )
posix_trace_eventid_equal( )
posix_trace_eventid_get_name( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_get_attr( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_rewind( )

In particular, notice that the operations normally used by a trace controller process, such as *posix_trace_start*( ), *posix_trace_stop*( ), or *posix_trace_shutdown*( ), cannot be invoked using the trace stream identifier returned by the *posix_trace_open*( ) function.

The *posix_trace_rewind*( ) function shall reset the current trace event timestamp, which specifies the timestamp of the trace event that will be read by the next call to *posix_trace_getnext_event*( ), to the timestamp of the oldest trace event recorded in the trace log identified by *trid*.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, the *posix_trace_open*( ) function stores the trace stream identifier value in the object pointed to by *trid*.

ERRORS

The *posix_trace_open*( ) function shall fail if:

[EINTR]	The operation was interrupted by a signal and thus no trace log was opened.
[EINVAL]	The object pointed to by *file_desc* does not correspond to a valid trace log.

The *posix_trace_close*( ) and *posix_trace_rewind*( ) functions may fail if:

[EINVAL]	The object pointed to by *trid* does not correspond to a valid trace log.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_get_attr( ), *posix_trace_get_filter*( ), *posix_trace_getnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

NAME

posix_trace_create, posix_trace_create_withlog, posix_trace_flush, posix_trace_shutdown — trace stream initialization, flush, and shutdown from a process (**TRACING**)

SYNOPSIS

```
TRC       #include <sys/types.h>
          #include <trace.h>

          int posix_trace_create(pid_t pid,
              const trace_attr_t *restrict attr,
              trace_id_t *restrict trid);
TRC  TRL  int posix_trace_create_withlog(pid_t pid,
              const trace_attr_t *restrict attr, int file_desc,
              trace_id_t *restrict trid);
          int posix_trace_flush(trace_id_t trid);
TRC       int posix_trace_shutdown(trace_id_t trid);
```

DESCRIPTION

The *posix_trace_create*( ) function shall create an active trace stream. It allocates all the resources needed by the trace stream being created for tracing the process specified by *pid* in accordance with the *attr* argument. The *attr* argument represents the initial attributes of the trace stream and shall have been initialized by the function *posix_trace_attr_init*( ) prior the *posix_trace_create*( ) call. If the argument *attr* is NULL, the default attributes shall be used. The *attr* attributes object shall be manipulated through a set of functions described in the *posix_trace_attr* family of functions. If the attributes of the object pointed to by *attr* are modified later, the attributes of the trace stream shall not be affected. The *creation-time* attribute of the newly created trace stream shall be set to the value of the system clock, if the Timers option is not supported, or to the value of the CLOCK_REALTIME clock, if the Timers option is supported.

The *pid* argument represents the target process to be traced. If the process executing this function does not have appropriate privileges to trace the process identified by *pid*, an error shall be returned. If the *pid* argument is zero, the calling process shall be traced.

The *posix_trace_create*( ) function shall store the trace stream identifier of the new trace stream in the object pointed to by the *trid* argument. This trace stream identifier shall be used in subsequent calls to control tracing. The *trid* argument may only be used by the following functions:

posix_trace_clear( )
posix_trace_eventid_equal( )
posix_trace_eventid_get_name( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_get_attr( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_shutdown( )
posix_trace_start( )
posix_trace_stop( )
posix_trace_timedgetnext_event( )
posix_trace_trid_eventid_open( )
posix_trace_trygetnext_event( )

TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid* argument:

posix_trace_get_filter( ) *posix_trace_set_filter*( )

In particular, notice that the operations normally used by a trace analyzer process, such as *posix_trace_rewind*( ) or *posix_trace_close*( ), cannot be invoked using the trace stream identifier returned by the *posix_trace_create*( ) function.

TEF A trace stream shall be created in a suspended state. If the Trace Event Filter option is supported, its trace event type filter shall be empty.

The *posix_trace_create*( ) function may be called multiple times from the same or different processes, with the system-wide limit indicated by the runtime invariant value {TRACE_SYS_MAX}, which has the minimum value {_POSIX_TRACE_SYS_MAX}.

The trace stream identifier returned by the *posix_trace_create*( ) function in the argument pointed to by *trid* is valid only in the process that made the function call. If it is used from another process, that is a child process, in functions defined in IEEE Std 1003.1-200x, these functions shall return with the error [EINVAL].

TRL The *posix_trace_create_withlog*( ) function shall be equivalent to *posix_trace_create*( ), except that it associates a trace log with this stream. The *file_desc* argument shall be the file descriptor designating the trace log destination. The function shall fail if this file descriptor refers to a file with a file type that is not compatible with the log policy associated with the trace log. The list of the appropriate file types that are compatible with each log policy is implementation-defined.

The *posix_trace_create_withlog*( ) function shall return in the parameter pointed to by *trid* the trace stream identifier, which uniquely identifies the newly created trace stream, and shall be used in subsequent calls to control tracing. The *trid* argument may only be used by the following functions:

posix_trace_clear( )
posix_trace_eventid_equal( )
posix_trace_eventid_get_name( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_flush( )
posix_trace_get_attr( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_shutdown( )
posix_trace_start( )
posix_trace_stop( )
posix_trace_timedgetnext_event( )
posix_trace_trid_eventid_open( )
posix_trace_trygetnext_event( )

TRL TEF If the Trace Event Filter option is supported, the following additional functions may use the *trid* argument:

posix_trace_get_filter( )
posix_trace_set_filter( )

TRL In particular, notice that the operations normally used by a trace analyzer process, such as *posix_trace_rewind*( ) or *posix_trace_close*( ), cannot be invoked using the trace stream identifier returned by the *posix_trace_create_withlog*( ) function.

The *posix_trace_flush*( ) function shall initiate a flush operation which copies the contents of the trace stream identified by the argument *trid* into the trace log associated with the trace stream at the creation time. If no trace log has been associated with the trace stream pointed to by *trid*, this function shall return an error. The termination of the flush operation can be polled by the *posix_trace_get_status*( ) function. During the flush operation, it shall be possible to trace new trace events up to the point when the trace stream becomes full. After flushing is completed, the space used by the flushed trace events shall be available for tracing new trace events.

If flushing the trace stream causes the resulting trace log to become full, the trace log full policy shall be applied. If the trace *log-full-policy* attribute is set, the following occurs:

POSIX_TRACE_UNTIL_FULL
The trace events that have not yet been flushed shall be discarded.

POSIX_TRACE_LOOP
The trace events that have not yet been flushed shall be written to the beginning of the trace log, overwriting previous trace events stored there.

POSIX_TRACE_APPEND
The trace events that had not yet been flushed shall be appended to the trace log.

The *posix_trace_shutdown*( ) function shall stop the tracing of trace events in the trace stream identified by *trid*, as if *posix_trace_stop*( ) had been invoked. The *posix_trace_shutdown*( ) function shall free all the resources associated with the trace stream.

The *posix_trace_shutdown*( ) function shall not return until all the resources associated with the trace stream have been freed. When the *posix_trace_shutdown*( ) function returns, the *trid* argument becomes an invalid trace stream identifier. A call to this function shall unconditionally deallocate the resources regardless of whether all trace events have been retrieved by the analyzer process. Any thread blocked on one of the *trace_getnext_event*( ) functions (which specified this *trid*) before this call is unblocked with the error [EINVAL].

If the process exits, invokes an *exec*( ) call, or is terminated, the trace streams that the process had created and that have not yet been shut down, shall be automatically shut down as if an explicit call were made to the *posix_trace_shutdown*( ) function.

TRL For an active trace stream with log, when the *posix_trace_shutdown*( ) function is called, all trace events that have not yet been flushed to the trace log shall be flushed, as in the *posix_trace_flush*( ) function, and the trace log shall be closed.

When a trace log is closed, all the information that may be retrieved later from the trace log through the trace interface shall have been written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.

In addition, unspecified information shall be written to the trace log to allow detection of a valid trace log during the *posix_trace_open*( ) operation.

The *posix_trace_shutdown*( ) function shall not return until all trace events have been flushed.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

TRL The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions store the trace stream identifier value in the object pointed to by *trid*, if successful.

ERRORS

TRL The *posix_trace_create*( ) and *posix_trace_create_withlog*( ) functions shall fail if:

[EAGAIN]	No more trace streams can be started now. {TRACE_SYS_MAX} has been exceeded.
[EINTR]	The operation was interrupted by a signal. No trace stream was created.
[EINVAL]	One or more of the trace parameters specified by the *attr* parameter is invalid.

[ENOMEM]	The implementation does not currently have sufficient memory to create the trace stream with the specified parameters.
[EPERM]	The caller does not have appropriate privilege to trace the process specified by *pid*.
[ESRCH]	The *pid* argument does not refer to an existing process.

TRL The *posix_trace_create_withlog*( ) function shall fail if:

[EBADF]	The *file_desc* argument is not a valid file descriptor open for writing.
[EINVAL]	The *file_desc* argument refers to a file with a file type that does not support the log policy associated with the trace log.
[ENOSPC]	No space left on device. The device corresponding to the argument *file_desc* does not contain the space required to create this trace log.

TRL The *posix_trace_flush*( ) and *posix_trace_shutdown*( ) functions shall fail if:

[EINVAL]	The value of the *trid* argument does not correspond to an active trace stream with log.
[EFBIG]	The trace log file has attempted to exceed an implementation-defined maximum file size.
[ENOSPC]	No space left on device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *exec*, *posix_trace_attr_init*( ), *posix_trace_clear*( ), *posix_trace_close*( ), *posix_trace_eventid_equal*( ), *posix_trace_eventtypelist_getnext_id*( ), *posix_trace_flush*( ), *posix_trace_get_attr*( ), *posix_trace_get_filter*( ), *posix_trace_get_status*( ), *posix_trace_getnext_event*( ), *posix_trace_open*( ), *posix_trace_rewind*( ), *posix_trace_set_filter*( ), *posix_trace_shutdown*( ), *posix_trace_start*( ), *posix_trace_timedgetnext_event*( ), *posix_trace_trid_eventid_open*( ), *posix_trace_start*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

NAME

posix_trace_event, posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>
```

```
void posix_trace_event(trace_event_id_t event_id,
    const void *restrictdata_ptr, size_t data_len);
int posix_trace_eventid_open(const char *restrict event_name,
    trace_event_id_t *restrict event_id);
```

DESCRIPTION

The *posix_trace_event*( ) function shall record the *event_id* and the user data pointed to by *data_ptr* in the trace stream into which the calling process is being traced and in which *event_id* is not filtered out. If the total size of the user trace event data represented by *data_len* is not greater than the declared maximum size for user trace event data, then the *truncation-status* attribute of the trace event recorded is POSIX_TRACE_NOT_TRUNCATED. Otherwise, the user trace event data is truncated to this declared maximum size and the *truncation-status* attribute of the trace event recorded is POSIX_TRACE_TRUNCATED_RECORD.

If there is no trace stream created for the process or if the created trace stream is not running or if the trace event specified by *event_id* is filtered out in the trace stream, the *posix_trace_event*( ) function shall have no effect.

The *posix_trace_eventid_open*( ) function shall associate a user trace event name with a trace event type identifier for the calling process. The trace event name is the string pointed to by the argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user trace event type identifiers that can be defined for any given process is limited by the maximum value {TRACE_USER_EVENT_MAX}, which has the minimum value {POSIX_TRACE_USER_EVENT_MAX}.

If the Trace Inherit option is not supported, the *posix_trace_eventid_open*( ) function shall associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier that is unique for the traced process, and is returned in the variable pointed to by the *event_id* argument. If the user trace event name has already been mapped for the traced process, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 529)) user trace event shall be returned.

TRI If the Trace Inherit option is supported, the *posix_trace_eventid_open*( ) function shall associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier that is unique for all the processes being traced in this same trace stream, and is returned in the variable pointed to by the *event_id* argument. If the user trace event name has already been mapped for the traced processes, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_USEREVENT (Table 2-7 (on page 529)) user trace event shall be returned.

Note: The above procedure, together with the fact that multiple processes can only be traced into the same trace stream by inheritance, ensure that all the processes that are traced into a trace stream have the same mapping of trace event names to trace event type identifiers.

If there is no trace stream created, the *posix_trace_eventid_open*( ) function shall store this information for future trace streams created for this process.

RETURN VALUE

No return value is defined for the *posix_trace_event*( ) function.

Upon successful completion, the *posix_trace_eventid_open*( ) function shall return a value of zero. Otherwise, it shall return the corresponding error number. The *posix_trace_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event_id*, if successful.

ERRORS

The *posix_trace_eventid_open*( ) function shall fail if: |

[ENAMETOOLONG]
The size of the name pointed to by *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_start( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

IEEE PASC Interpretation 1003.1 #123 is applied. |

IEEE PASC Interpretation 1003.1 #127 is applied, correcting some editorial errors in the names of the *posix_trace_eventid_open*( ) function and the *event_id* argument. |

NAME

posix_trace_eventid_equal, posix_trace_eventid_get_name, posix_trace_trid_eventid_open — manipulate trace event type identifier (**TRACING**)

SYNOPSIS

```
#include <trace.h>

int posix_trace_eventid_equal(trace_id_t trid, trace_event_id_t event1,
    trace_event_id_t event2);
int posix_trace_eventid_get_name(trace_id_t trid,
    trace_event_id_t event, char *event_name);
int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *restrict event_name,
    trace_event_id_t *restrict event);
```

DESCRIPTION

The *posix_trace_eventid_equal*( ) function shall compare the trace event type identifiers *event1* and *event2* from the same trace stream or the same trace log identified by the *trid* argument. If the trace event type identifiers *event1* and *event2* are from different trace streams, the return value shall be unspecified.

The *posix_trace_eventid_get_name*( ) function shall return in the argument pointed to by *event_name*, the trace event name associated with the trace event type identifier identified by the argument *event*, for the trace stream or for the trace log identified by the *trid* argument. The name of the trace event shall have a maximum of {TRACE_EVENT_NAME_MAX} characters (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). Successive calls to this function with the same trace event type identifier and the same trace stream identifier shall return the same event name.

The *posix_trace_trid_eventid_open*( ) function shall associate a user trace event name with a trace event type identifier for a given trace stream. The trace stream is identified by the *trid* argument, and it shall be an active trace stream. The trace event name is the string pointed to by the argument *event_name*. It shall have a maximum of {TRACE_EVENT_NAME_MAX} characters (which has the minimum value {_POSIX_TRACE_EVENT_NAME_MAX}). The number of user trace event type identifiers that can be defined for any given process is limited by the maximum value {TRACE_USER_EVENT_MAX}, which has the minimum value {_POSIX_TRACE_USER_EVENT_MAX}.

If the Trace Inherit option is not supported, the *posix_trace_trid_eventid_open*( ) function shall associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier that is unique for the process being traced in the trace stream identified by the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user trace event name has already been mapped for the traced process, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 529)) user trace event shall be returned.

If the Trace Inherit option is supported, the *posix_trace_trid_eventid_open*( ) function shall associate the user trace event name pointed to by the *event_name* argument with a trace event type identifier that is unique for all the processes being traced in the trace stream identified by the *trid* argument, and is returned in the variable pointed to by the *event* argument. If the user trace event name has already been mapped for the traced processes, then the previously assigned trace event type identifier shall be returned. If the per-process user trace event name limit represented by {TRACE_USER_EVENT_MAX} has been reached, the pre-defined

POSIX_TRACE_UNNAMED_USEREVENT (see Table 2-7 (on page 529)) user trace event shall be returned.

RETURN VALUE

TEF Upon successful completion, the *posix_trace_eventid_get_name*( ) and *posix_trace_trid_eventid_open*( ) functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The *posix_trace_eventid_equal*( ) function shall return a non-zero value if *event1* and *event2* are equal; otherwise, a value of zero shall be returned. No errors are defined. If either *event1* or *event2* are not valid trace event type identifiers for the trace stream specified by *trid* or if the *trid* is invalid, the behavior shall be unspecified.

The *posix_trace_eventid_get_name*( ) function stores the trace event name value in the object pointed to by *event_name*, if successful.

TEF The *posix_trace_trid_eventid_open*( ) function stores the trace event type identifier value in the object pointed to by *event*, if successful.

ERRORS

TEF The *posix_trace_eventid_get_name*( ) and *posix_trace_trid_eventid_open*( ) functions shall fail if:

[EINVAL] The *trid* argument was not a valid trace stream identifier.

TEF The *posix_trace_trid_eventid_open*( ) function shall fail if:

TEF [ENAMETOOLONG]
The size of the name pointed to by *event_name* argument was longer than the implementation-defined value {TRACE_EVENT_NAME_MAX}.

The *posix_trace_eventid_get_name*( ) function shall fail if:

[EINVAL] The trace event type identifier *event* was not associated with any name.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_event( ), *posix_trace_getnext_event*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

NAME

posix_trace_eventid_get_name — manipulate trace event type identifier (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_eventid_get_name(trace_id_t trid,
    trace_event_id_t event, char *event_name);
```

DESCRIPTION

Refer to *posix_trace_eventid_equal*( ).

NAME

posix_trace_eventid_open — trace functions for instrumenting application code (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_eventid_open(const char *restrict event_name,
    trace_event_id_t *restrict event_id);
```

DESCRIPTION

Refer to *posix_trace_event*( ).

NAME

posix_trace_eventset_add, posix_trace_eventset_del, posix_trace_eventset_empty, posix_trace_eventset_fill, posix_trace_eventset_ismember — manipulate trace event type sets (**TRACING**)

SYNOPSIS

TRC TEF

```
#include <trace.h>

int posix_trace_eventset_add(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_del(trace_event_id_t event_id,
    trace_event_set_t *set);
int posix_trace_eventset_empty(trace_event_set_t *set);
int posix_trace_eventset_fill(trace_event_set_t *set, int what);
int posix_trace_eventset_ismember(trace_event_id_t event_id,
    const trace_event_set_t *restrict set,
    int *restrict ismember);
```

DESCRIPTION

These primitives manipulate sets of trace event types. They operate on data objects addressable by the application, not on the current trace event filter of any trace stream.

The *posix_trace_eventset_add*( ) and *posix_trace_eventset_del*( ) functions, respectively, shall add or delete the individual trace event type specified by the value of the argument *event_id* to or from the trace event type set pointed to by the argument *set*. Adding a trace event type already in the set or deleting a trace event type not in the set shall not be considered an error.

The *posix_trace_eventset_empty*( ) function shall initialize the trace event type set pointed to by the *set* argument such that all trace event types defined, both system and user, shall be excluded from the set.

The *posix_trace_eventset_fill*( ) function shall initialize the trace event type set pointed to by the argument *set*, such that the set of trace event types defined by the argument *what* shall be included in the set. The value of the argument *what* shall consist of one of the following values, as defined in the **<trace.h>** header:

POSIX_TRACE_WOPID_EVENTS
All the process-independent implementation-defined system trace event types are included in the set.

POSIX_TRACE_SYSTEM_EVENTS All the implementation-defined system trace event types are included in the set, as are those defined in IEEE Std 1003.1-200x.

POSIX_TRACE_ALL_EVENTS All trace event types defined, both system and user, are included in the set.

Applications shall call either *posix_trace_eventset_empty*( ) or *posix_trace_eventset_fill*( ) at least once for each object of type **trace_event_set_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of the *posix_trace_eventset_add*( ), *posix_trace_eventset_del*( ), or *posix_trace_eventset_ismember*( ) functions, the results are undefined.

The *posix_trace_eventset_ismember*( ) function shall test whether the trace event type specified by the value of the argument *event_id* is a member of the set pointed to by the argument *set*. The value returned in the object pointed to by *ismember* argument is zero if the trace event type identifier is not a member of the set and a value different from zero if it is a member of the set. |

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

ERRORS

These functions may fail if:

[EINVAL] The value of one of the arguments is invalid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_set_filter( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

NAME

posix_trace_eventtypelist_getnext_id, posix_trace_eventtypelist_rewind — iterate over a mapping of trace event types (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_eventtypelist_getnext_id(trace_id_t trid,
    trace_event_id_t *restrict event, int *restrict unavailable);
int posix_trace_eventtypelist_rewind(trace_id_t trid);
```

DESCRIPTION

The first time *posix_trace_eventtypelist_getnext_id*( ) is called, the function shall return in the variable pointed to by *event* the first trace event type identifier of the list of trace events of the trace stream identified by the *trid* argument. Successive calls to *posix_trace_eventtypelist_getnext_id*( ) return in the variable pointed to by *event* the next trace event type identifier in that same list. Each time a trace event type identifier is successfully written into the variable pointed to by the *event* argument, the variable pointed to by the *unavailable* argument shall be set to zero. When no more trace event type identifiers are available, and so none is returned, the variable pointed to by the *unavailable* argument shall be set to a value different from zero.

The *posix_trace_eventtypelist_rewind*( ) function shall reset the next trace event type identifier to be read to the first trace event type identifier from the list of trace events used in the trace stream identified by *trid*.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The *posix_trace_eventtypelist_getnext_id*( ) function stores the trace event type identifier value in the object pointed to by *event*, if successful.

ERRORS

These functions shall fail if:

[EINVAL] The *trid* argument was not a valid trace stream identifier.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_event( ), *posix_trace_getnext_event*( ), *posix_trace_trid_eventid_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretations 1003.1 #123 and #129 are applied.

NAME

posix_trace_flush — trace stream flush from a process (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_flush(trace_id_t trid);
```

DESCRIPTION

Refer to *posix_trace_create*( ).

NAME

posix_trace_get_attr, posix_trace_get_status — retrieve the trace attributes or trace statuses (**TRACING**)

SYNOPSIS

TRC

```
#include <trace.h>

int posix_trace_get_attr(trace_id_t trid, trace_attr_t *attr);
int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```

DESCRIPTION

The *posix_trace_get_attr*( ) function shall copy the attributes of the active trace stream identified by *trid* into the object pointed to by the *attr* argument. TRL If the Trace Log option is supported, *trid* may represent a pre-recorded trace log.

The *posix_trace_get_status*( ) function shall return, in the structure pointed to by the *statusinfo* argument, the current trace status for the trace stream identified by the *trid* argument. These status values returned in the structure pointed to by *statusinfo* shall have been appropriately read to ensure that the returned values are consistent. TRL If the Trace Log option is supported and the *trid* argument refers to a pre-recorded trace stream, the status shall be the status of the completed trace stream.

Each time the *posix_trace_get_status*( ) function is used, the overrun status of the trace stream shall be reset to POSIX_TRACE_NO_OVERRUN immediately after the call completes. TRL If the Trace Log option is supported, the *posix_trace_get_status*( ) function shall behave the same as when the option is not supported except for the following differences:

- If the *trid* argument refers to a trace stream with log, each time the *posix_trace_get_status*( ) function is used, the log overrun status of the trace stream shall be reset to POSIX_TRACE_NO_OVERRUN and the *flush_error* status shall be reset to zero immediately after the call completes.
- If the *trid* argument refers to a pre-recorded trace stream, the status returned shall be the status of the completed trace stream and the status values of the trace stream shall not be reset.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The *posix_trace_get_attr*( ) function stores the trace attributes in the object pointed to by *attr*, if successful.

The *posix_trace_get_status*( ) function stores the trace status in the object pointed to by *statusinfo*, if successful.

ERRORS

These functions shall fail if:

[EINVAL] The trace stream argument *trid* does not correspond to a valid active trace stream or a valid trace log.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_attr_destroy( ), *posix_trace_attr_init*( ), *posix_trace_create*( ), *posix_trace_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

NAME

posix_trace_get_filter, posix_trace_set_filter — retrieve and set filter of an initialized trace stream (**TRACING**)

SYNOPSIS

TRC TEF

```
#include <trace.h>

int posix_trace_get_filter(trace_id_t trid, trace_event_set_t *set);
int posix_trace_set_filter(trace_id_t trid,
    const trace_event_set_t *set, int how);
```

DESCRIPTION

The *posix_trace_get_filter*( ) function shall retrieve, into the argument pointed to by *set*, the actual trace event filter from the trace stream specified by *trid*.

The *posix_trace_set_filter*( ) function shall change the set of filtered trace event types after a trace stream identified by the *trid* argument is created. This function may be called prior to starting the trace stream, or while the trace stream is active. By default, if no call is made to *posix_trace_set_filter*( ), all trace events shall be recorded (that is, none of the trace event types are filtered out).

If this function is called while the trace is in progress, a special system trace event, POSIX_TRACE_FILTER, shall be recorded in the trace indicating both the old and the new sets of filtered trace event types (see Table 2-4 (on page 528) and Table 2-6 (on page 529)).

If the *posix_trace_set_filter*( ) function is interrupted by a signal, an error shall be returned and the filter shall not be changed. In this case, the state of the trace stream shall not be changed.

The value of the argument *how* indicates the manner in which the set is to be changed and shall have one of the following values, as defined in the **<trace.h>** header:

POSIX_TRACE_SET_EVENTSET
The resulting set of trace event types to be filtered shall be the trace event type set pointed to by the argument *set*.

POSIX_TRACE_ADD_EVENTSET
The resulting set of trace event types to be filtered shall be the union of the current set and the trace event type set pointed to by the argument *set*.

POSIX_TRACE_SUB_EVENTSET
The resulting set of trace event types to be filtered shall be all trace event types in the current set that are not in the set pointed to by the argument *set*; that is, remove each element of the specified set from the current filter.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

The *posix_trace_get_filter*( ) function stores the set of filtered trace event types in *set*, if successful.

ERRORS

These functions shall fail if:

[EINVAL]	The value of the *trid* argument does not correspond to an active trace stream or the value of the argument pointed to by *set* is invalid.
[EINTR]	The operation was interrupted by a signal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_eventset_add( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #123 is applied.

NAME

posix_trace_get_status — retrieve the trace statuses (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>

int posix_trace_get_status(trace_id_t trid,
    struct posix_trace_status_info *statusinfo);
```

DESCRIPTION

Refer to *posix_trace_get_attr*( ).

NAME

posix_trace_getnext_event, posix_trace_timedgetnext_event, posix_trace_trygetnext_event — retrieve a trace event (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>
```

```
int posix_trace_getnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable);
```

TRC TMO
```
int posix_trace_timedgetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable,
    const struct timespec *restrict abs_timeout);
```

TRC
```
int posix_trace_trygetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable);
```

DESCRIPTION

The *posix_trace_getnext_event*( ) function shall report a recorded trace event either from an active trace stream without log TRL or a pre-recorded trace stream identified by the *trid* argument. The *posix_trace_trygetnext_event*( ) function shall report a recorded trace event from an active trace stream without log identified by the *trid* argument.

The trace event information associated with the recorded trace event shall be copied by the function into the structure pointed to by the argument *event* and the data associated with the trace event shall be copied into the buffer pointed to by the *data* argument.

The *posix_trace_getnext_event*( ) function shall block if the *trid* argument identifies an active trace stream and there is currently no trace event ready to be retrieved. When returning, if a recorded trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, the variable pointed to by the *unavailable* argument shall be set to a value different from zero.

TMO The *posix_trace_timedgetnext_event*( ) function shall attempt to get another trace event from an active trace stream without log, as in the *posix_trace_getnext_event*( ) function. However, if no trace event is available from the trace stream, the implied wait shall be terminated when the timeout specified by the argument *abs_timeout* expires, and the function shall return the error [ETIMEDOUT].

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock upon which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already passed at the time of the call.

TMO TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

TMO Under no circumstance shall the function fail with a timeout if a trace event is immediately available from the trace stream. The validity of the *abs_timeout* argument need not be checked if a trace event is immediately available from the trace stream.

The behavior of this function for a pre-recorded trace stream is unspecified.

TRL The *posix_trace_trygetnext_event*( ) function shall not block. This function shall return an error if the *trid* argument identifies a pre-recorded trace stream. If a recorded trace event was reported, the variable pointed to by the *unavailable* argument shall be set to zero. Otherwise, if no trace event was reported, the variable pointed to by the *unavailable* argument shall be set to a value different from zero.

The argument *num_bytes* shall be the size of the buffer pointed to by the *data* argument. The argument *data_len* reports to the application the length in bytes of the data record just transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace event pointed to by the *event* argument, all the recorded data shall be transferred. In this case, the *truncation-status* member of the trace event structure shall be either POSIX_TRACE_NOT_TRUNCATED, if the trace event data was recorded without truncation while tracing, or POSIX_TRACE_TRUNCATED_RECORD, if the trace event data was truncated when it was recorded. If the *num_bytes* argument is less than the length of recorded trace event data, the data transferred shall be truncated to a length of *num_bytes*, the value stored in the variable pointed to by *data_len* shall be equal to *num_bytes*, and the *truncation-status* member of the *event* structure argument shall be set to POSIX_TRACE_TRUNCATED_READ (see the *posix_trace_event_info*( ) function).

The report of a trace event shall be sequential starting from the oldest recorded trace event. Trace events shall be reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events occurring very close to each other to be unknown. Once reported, a trace event cannot be reported again from an active trace stream. Once a trace event is reported from an active trace stream without log, the trace stream shall make the resources associated with that trace event available to record future generated trace events.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

If successful, these functions store:

- The recorded trace event in the object pointed to by *event*
- The trace event information associated with the recorded trace event in the object pointed to by *data*
- The length of this trace event information in the object pointed to by *data_len*
- The value of zero in the object pointed to by *unavailable*

ERRORS

These functions shall fail if: |

[EINVAL] The trace stream identifier argument *trid* is invalid.

The *posix_trace_getnext_event*( ) and *posix_trace_timedgetnext_event*( ) functions shall fail if: |

[EINTR] The operation was interrupted by a signal, and so the call had no effect.

The *posix_trace_trygetnext_event*( ) function shall fail if: |

[EINVAL] The trace stream identifier argument *trid* does not correspond to an active trace stream.

TMO The *posix_trace_timedgetnext_event*( ) function shall fail if:

[EINVAL]	There is no trace event immediately available from the trace stream, and the *timeout* argument is invalid.
[ETIMEDOUT]	No trace event was available from the trace stream before the specified timeout *timeout* expired.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_create( ), **posix_trace_event_info Structure**, *posix_trace_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

IEEE PASC Interpretation 1003.1 #123 is applied. |

NAME

posix_trace_open — trace log management (**TRACING**)

SYNOPSIS

TCT TRL

```
#include <trace.h>

int posix_trace_open(int file_desc, trace_id_t *trid);
```

DESCRIPTION

Refer to *posix_trace_close*( ).

NAME

posix_trace_rewind — trace log management (**TRACING**)

SYNOPSIS

TCT TRL

```
#include <trace.h>

int posix_trace_rewind(trace_id_t trid);
```

DESCRIPTION

Refer to *posix_trace_close*( ).

NAME

posix_trace_set_filter — set filter of an initialized trace stream (**TRACING**)

SYNOPSIS

TRC TEF

```
#include <trace.h>

int posix_trace_set_filter(trace_id_t trid,
    const trace_event_set_t *set, int how);
```

DESCRIPTION

Refer to *posix_trace_get_filter*( ).

NAME

posix_trace_shutdown — trace stream shutdown from a process (**TRACING**)

SYNOPSIS

TRC
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_shutdown(trace_id_t trid);
```

DESCRIPTION

Refer to *posix_trace_create*( ).

NAME

posix_trace_start, posix_trace_stop — trace start and stop (**TRACING**)

SYNOPSIS

TRC
```
#include <trace.h>

int posix_trace_start(trace_id_t trid);
int posix_trace_stop (trace_id_t trid);
```

DESCRIPTION

The *posix_trace_start*( ) and *posix_trace_stop*( ) functions, respectively, shall start and stop the trace stream identified by the argument *trid*.

The effect of calling the *posix_trace_start*( ) function shall be recorded in the trace stream as the POSIX_TRACE_START system trace event and the status of the trace stream shall become POSIX_TRACE_RUNNING. If the trace stream is in progress when this function is called, the POSIX_TRACE_START system trace event shall not be recorded and the trace stream shall continue to run. If the trace stream is full, the POSIX_TRACE_START system trace event shall not be recorded and the status of the trace stream shall not be changed.

The effect of calling the *posix_trace_stop*( ) function shall be recorded in the trace stream as the POSIX_TRACE_STOP system trace event and the status of the trace stream shall become POSIX_TRACE_SUSPENDED. If the trace stream is suspended when this function is called, the POSIX_TRACE_STOP system trace event shall not be recorded and the trace stream shall remain suspended. If the trace stream is full, the POSIX_TRACE_STOP system trace event shall not be recorded and the status of the trace stream shall not be changed.

RETURN VALUE

Upon successful completion, these functions shall return a value of zero. Otherwise, they shall return the corresponding error number.

ERRORS

These functions shall fail if:

[EINVAL]	The value of the argument *trid* does not correspond to an active trace stream and thus no trace stream was started or stopped.
[EINTR]	The operation was interrupted by a signal and thus the trace stream was not necessarily started or stopped.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

posix_trace_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<trace.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1q-2000. |

IEEE PASC Interpretation 1003.1 #123 is applied. |

NAME

posix_trace_timedgetnext_event, — retrieve a trace event (**TRACING**)

SYNOPSIS

TRC TMO
```
#include <sys/types.h>
#include <trace.h>

int posix_trace_timedgetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

Refer to *posix_trace_getnext_event*( ).

NAME

posix_trace_trid_eventid_open — manipulate trace event type identifier (**TRACING**)

SYNOPSIS

TRC TEF
```
#include <trace.h>

int posix_trace_trid_eventid_open(trace_id_t trid,
    const char *restrict event_name,
    trace_event_id_t *restrict event);
```

DESCRIPTION

Refer to *posix_trace_eventid_equal*( ).

NAME

posix_trace_trygetnext_event — retrieve a trace event (**TRACING**)

SYNOPSIS

TRC

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_trygetnext_event(trace_id_t trid,
    struct posix_trace_event_info *restrict event,
    void *restrict data, size_t num_bytes,
    size_t *restrict data_len, int *restrict unavailable);
```

DESCRIPTION

Refer to *posix_trace_getnext_event*( ).

NAME

posix_typed_mem_get_info — query typed memory information (**ADVANCED REALTIME**)

SYNOPSIS

TYM

```
#include <sys/mman.h>

int posix_typed_mem_get_info(int fildes,
    struct posix_typed_mem_info *info);

```

DESCRIPTION

The *posix_typed_mem_get_info*( ) function shall return, in the *posix_tmi_length* field of the **posix_typed_mem_info** structure pointed to by *info*, the maximum length which may be successfully allocated by the typed memory object designated by *fildes*. This maximum length shall take into account the flag POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG specified when the typed memory object represented by *fildes* was opened. The maximum length is dynamic; therefore, the value returned is valid only while the current mapping of the corresponding typed memory pool remains unchanged.

If *fildes* represents a typed memory object opened with neither the POSIX_TYPED_MEM_ALLOCATE flag nor the POSIX_TYPED_MEM_ALLOCATE_CONTIG flag specified, the returned value of *info*->*posix_tmi_length* is unspecified.

The *posix_typed_mem_get_info*( ) function may return additional implementation-defined information in other fields of the **posix_typed_mem_info** structure pointed to by *info*.

If the memory object specified by *fildes* is not a typed memory object, then the behavior of this function is undefined.

RETURN VALUE

Upon successful completion, the *posix_typed_mem_get_info*( ) function shall return zero; otherwise, the corresponding error status value shall be returned.

ERRORS

The *posix_typed_mem_get_info*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid open file descriptor.
[ENODEV]	The *fildes* argument is not connected to a memory object supported by this function.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

An application that needs to allocate a block of typed memory with length dependent upon the amount of memory currently available must either query the typed memory object to obtain the amount available, or repeatedly invoke *mmap*( ) attempting to guess an appropriate length. While the latter method is existing practice with *malloc*( ), it is awkward and imprecise. The *posix_typed_mem_get_info*( ) function allows an application to immediately determine available memory. This is particularly important for typed memory objects that may in some cases be scarce resources. Note that when a typed memory pool is a shared resource, some form of mutual exclusion or synchronization may be required while typed memory is being queried and allocated to prevent race conditions.

The existing *fstat*( ) function is not suitable for this purpose. We realize that implementations may wish to provide other attributes of typed memory objects (for example, alignment requirements, page size, and so on). The *fstat*( ) function returns a structure which is not extensible and, furthermore, contains substantial information that is inappropriate for typed memory objects.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *mmap*( ), *posix_typed_mem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

posix_typed_mem_open — open a typed memory object (**ADVANCED REALTIME**)

SYNOPSIS

TYM

```
#include <sys/mman.h>

int posix_typed_mem_open(const char *name, int oflag, int tflag);
```

DESCRIPTION

The *posix_typed_mem_open*( ) function shall establish a connection between the typed memory object specified by the string pointed to by *name* and a file descriptor. It shall create an open file description that refers to the typed memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that typed memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument shall conform to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *posix_typed_mem_open*( ) with the same value of *name* shall refer to the same typed memory object. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

Each typed memory object supported in a system shall be identified by a name which specifies not only its associated typed memory pool, but also the path or port by which it is accessed. That is, the same typed memory pool accessed via several different ports shall have several different corresponding names. The binding between names and typed memory objects is established in an implementation-defined manner. Unlike shared memory objects, there is no way within IEEE Std 1003.1-200x for a program to create a typed memory object.

The value of *tflag* shall determine how the typed memory object behaves when subsequently mapped by calls to *mmap*( ). At most, one of the following flags defined in **<sys/mman.h>** may be specified:

POSIX_TYPED_MEM_ALLOCATE
Allocate on *mmap*( ).

POSIX_TYPED_MEM_ALLOCATE_CONTIG
Allocate contiguously on *mmap*( ).

POSIX_TYPED_MEM_MAP_ALLOCATABLE
Map on *mmap*( ), without affecting allocatability.

If *tflag* has the flag POSIX_TYPED_MEM_ALLOCATE specified, any subsequent call to *mmap*( ) using the returned file descriptor shall result in allocation and mapping of typed memory from the specified typed memory pool. The allocated memory may be a contiguous previously unallocated area of the typed memory pool or several non-contiguous previously unallocated areas (mapped to a contiguous portion of the process address space). If *tflag* has the flag POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*( ) using the returned file descriptor shall result in allocation and mapping of a single contiguous previously unallocated area of the typed memory pool (also mapped to a contiguous portion of the process address space). If *tflag* has none of the flags POSIX_TYPED_MEM_ALLOCATE or POSIX_TYPED_MEM_ALLOCATE_CONTIG specified, any subsequent call to *mmap*( ) using the returned file descriptor shall map an application-chosen area from the specified typed memory pool such that this mapped area becomes unavailable for allocation until unmapped by all processes. If *tflag* has the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE specified, any subsequent call to *mmap*( ) using the returned file descriptor shall map an application-chosen area from the specified typed memory pool without an effect on the availability of that area for allocation; that is, mapping such an object leaves each byte of the mapped area unallocated if it was unallocated prior to the mapping or allocated if it was allocated prior to the mapping. The appropriate privilege to specify the POSIX_TYPED_MEM_MAP_ALLOCATABLE flag is implementation-defined.

If successful, *posix_typed_mem_open*( ) shall return a file descriptor for the typed memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor shall not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor shall be cleared.

The behavior of *msync*( ), *ftruncate*( ), and all file operations other than *mmap*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *fstat*( ), *dup*( ), *dup2*( ), and *close*( ), is unspecified when passed a file descriptor connected to a typed memory object by this function.

The file status flags of the open file description shall be set according to the value of *oflag*. Applications shall specify exactly one of the three access mode values described below and defined in the **<fcntl.h>** header, as the value of *oflag*.

O_RDONLY	Open for read access only.
O_WRONLY	Open for write access only.
O_RDWR	Open for read or write access.

RETURN VALUE

Upon successful completion, the *posix_typed_mem_open*( ) function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *posix_typed_mem_open*( ) function shall fail if:

[EACCES]	The typed memory object exists and the permissions specified by *oflag* are denied.
[EINTR]	The *posix_typed_mem_open*( ) operation was interrupted by a signal.
[EINVAL]	The flags specified in *tflag* are invalid (more than one of POSIX_TYPED_MEM_ALLOCATE, POSIX_TYPED_MEM_ALLOCATE_CONTIG, or POSIX_TYPED_MEM_MAP_ALLOCATABLE is specified).
[EMFILE]	Too many file descriptors are currently in use by this process.
[ENAMETOOLONG]	The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENFILE]	Too many file descriptors are currently open in the system.
[ENOENT]	The named typed memory object does not exist.
[EPERM]	The caller lacks the appropriate privilege to specify the flag POSIX_TYPED_MEM_MAP_ALLOCATABLE in argument *tflag*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *dup*( ), *exec*, *fcntl*( ), *fstat*( ), *ftruncate*( ), *mmap*( ), *msync*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h,> <sys/mman.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

pow, powf, powl — power function

SYNOPSIS

```
#include <math.h>

double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the value of x raised to the power y, x^y. If x is negative, the application shall ensure that y is an integer value.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the value of x raised to the power y.

MX For finite values of $x < 0$, and finite non-integer values of y, a domain error shall occur and either a NaN (if representable), or an implementation-defined value shall be returned.

If the correct value would cause overflow, a range error shall occur and *pow*( ), *powf*( ), and *powl*( ) shall return HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

If the correct value would cause underflow, and is not representable, a range error may occur,
MX and either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If x or y is a NaN, a NaN shall be returned (unless specified elsewhere in this description).

For any value of y (including NaN), if x is +1, 1.0 shall be returned.

For any value of x (including NaN), if y is ±0, 1.0 shall be returned.

For any odd integer value of $y > 0$, if x is ±0, ±0 shall be returned.

For $y > 0$ and not an odd integer, if x is ±0, +0 shall be returned.

If x is –1, and y is ±Inf, 1.0 shall be returned.

For $|x| < 1$, if y is –Inf, +Inf shall be returned.

For $|x| > 1$, if y is –Inf, +0 shall be returned.

For $|x| < 1$, if y is +Inf, +0 shall be returned.

For $|x| > 1$, if y is +Inf, +Inf shall be returned.

For y an odd integer < 0, if x is –Inf, –0 shall be returned.

For $y < 0$ and not an odd integer, if x is –Inf, +0 shall be returned.

For y an odd integer > 0, if x is –Inf, –Inf shall be returned.

For $y > 0$ and not an odd integer, if x is –Inf, +Inf shall be returned.

For $y < 0$, if x is +Inf, +0 shall be returned.

For $y > 0$, if x is +Inf, +Inf shall be returned.

For y an odd integer < 0, if x is ±0, a pole error shall occur and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL shall be returned for *pow*( ), *powf*( ), and *powl*( ), respectively.

For $y < 0$ and not an odd integer, if x is ±0, a pole error shall occur and HUGE_VAL, HUGE_VALF, and HUGE_VALL shall be returned for *pow*( ), *powf*( ), and *powl*( ), respectively.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

Domain Error	The value of x is negative and y is a finite non-integer. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
MX Pole Error	The value of x is zero and y is negative. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.
Range Error	The result overflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exp( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *powf*( ) and *powl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

pread — read from a file

SYNOPSIS

XSI
```
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

DESCRIPTION

Refer to *read*( ).

NAME

printf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf(const char *restrict format, ...);
```

DESCRIPTION

Refer to *fprintf*( ).

NAME

pselect, select — synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/select.h>

int pselect(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    const struct timespec *restrict timeout,
    const sigset_t *restrict sigmask);
int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

DESCRIPTION

The *pselect*( ) function shall examine the file descriptor sets whose addresses are passed in the *readfds*, *writefds*, and *errorfds* parameters to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively.

The *select*( ) function shall be equivalent to the *pselect*( ) function, except as follows:

- For the *select*( ) function, the timeout period is given in seconds and microseconds in an argument of type **struct timeval**, whereas for the *pselect*( ) function the timeout period is given in seconds and nanoseconds in an argument of type **struct timespec**.
- The *select*( ) function has no *sigmask* argument; it shall behave as *pselect*( ) does when *sigmask* is a null pointer.
- Upon successful completion, the *select*( ) function may modify the object pointed to by the *timeout* argument.

The *pselect*( ) and *select*( ) functions shall support regular files, terminal and pseudo-terminal devices, XSR STREAMS-based files, FIFOs, pipes, and sockets. The behavior of *pselect*( ) and *select*( ) on file descriptors that refer to other types of file is unspecified.

The *nfds* argument specifies the range of descriptors to be tested. The first *nfds* descriptors shall be checked in each set; that is, the descriptors from zero through *nfds*−1 in the descriptor sets shall be examined.

If the *readfds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for being ready to read, and on output indicates which file descriptors are ready to read.

If the *writefds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for being ready to write, and on output indicates which file descriptors are ready to write.

If the *errorfds* argument is not a null pointer, it points to an object of type **fd_set** that on input specifies the file descriptors to be checked for error conditions pending, and on output indicates which file descriptors have error conditions pending.

Upon successful completion, the *pselect*( ) or *select*( ) function shall modify the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments to indicate which file descriptors are ready for reading, ready for writing, or have an error condition pending, respectively, and shall return the total number of ready descriptors in all the output sets. For each file descriptor less than *nfds*, the corresponding bit shall be set on successful completion if it was set on input and the associated condition is true for that file descriptor.

If none of the selected descriptors are ready for the requested operation, the *pselect*( ) or *select*( ) function shall block until at least one of the requested operations becomes ready, until the *timeout* occurs, or until interrupted by a signal. The *timeout* parameter controls how long the *pselect*( ) or *select*( ) function shall take before timing out. If the *timeout* parameter is not a null pointer, it specifies a maximum interval to wait for the selection to complete. If the specified time interval expires without any requested operation becoming ready, the function shall return. If the *timeout* parameter is a null pointer, then the call to *pselect*( ) or *select*( ) shall block indefinitely until at least one descriptor meets the specified criteria. To effect a poll, the *timeout* parameter should not be a null pointer, and should point to a zero-valued **timespec** structure.

The use of a timeout does not affect any pending timers set up by *alarm*( ), *ualarm*( ), or *setitimer*( ).

Implementations may place limitations on the maximum timeout interval supported. All implementations shall support a maximum timeout interval of at least 31 days. If the *timeout* argument specifies a timeout interval greater than the implementation-defined maximum value, the maximum value shall be used as the actual timeout value. Implementations may also place limitations on the granularity of timeout intervals. If the requested timeout interval requires a finer granularity than the implementation supports, the actual timeout interval shall be rounded up to the next supported value.

If *sigmask* is not a null pointer, then the *pselect*( ) function shall replace the signal mask of the process by the set of signals pointed to by *sigmask* before examining the descriptors, and shall restore the signal mask of the process before returning.

A descriptor shall be considered ready for reading when a call to an input function with O_NONBLOCK clear would not block, whether or not the function would transfer data successfully. (The function might return data, an end-of-file indication, or an error other than one indicating that it is blocked, and in each of these cases the descriptor shall be considered ready for reading.)

A descriptor shall be considered ready for writing when a call to an output function with O_NONBLOCK clear would not block, whether or not the function would transfer data successfully.

If a socket has a pending error, it shall be considered to have an exceptional condition pending. Otherwise, what constitutes an exceptional condition is file type-specific. For a file descriptor for use with a socket, it is protocol-specific except as noted below. For other file types it is implementation-defined. If the operation is meaningless for a particular file type, *pselect*( ) or *select*( ) shall indicate that the descriptor is ready for read or write operations, and shall indicate that the descriptor has no exceptional condition pending.

If a descriptor refers to a socket, the implied input function is the *recvmsg*( ) function with parameters requesting normal and ancillary data, such that the presence of either type shall cause the socket to be marked as readable. The presence of out-of-band data shall be checked if the socket option SO_OOBINLINE has been enabled, as out-of-band data is enqueued with normal data. If the socket is currently listening, then it shall be marked as readable if an incoming connection request has been received, and a call to the *accept*( ) function shall complete without blocking.

If a descriptor refers to a socket, the implied output function is the *sendmsg*( ) function supplying an amount of normal data equal to the current value of the SO_SNDLOWAT option for the socket. If a non-blocking call to the *connect*( ) function has been made for a socket, and the connection attempt has either succeeded or failed leaving a pending error, the socket shall be marked as writable.

A socket shall be considered to have an exceptional condition pending if a receive operation with O_NONBLOCK clear for the open file description and with the MSG_OOB flag set would return out-of-band data without blocking. (It is protocol-specific whether the MSG_OOB flag would be used to read out-of-band data.) A socket shall also be considered to have an exceptional condition pending if an out-of-band data mark is present in the receive queue. Other circumstances under which a socket may be considered to have an exceptional condition pending are protocol-specific and implementation-defined.

If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is not a null pointer, the *pselect*( ) or *select*( ) function shall block for the time specified, or until interrupted by a signal. If the *readfds*, *writefds*, and *errorfds* arguments are all null pointers and the *timeout* argument is a null pointer, the *pselect*( ) or *select*( ) function shall block until interrupted by a signal.

File descriptors associated with regular files shall always select true for ready to read, ready to write, and error conditions.

On failure, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments shall not be modified. If the timeout interval expires without the specified condition being true for any of the specified file descriptors, the objects pointed to by the *readfds*, *writefds*, and *errorfds* arguments shall have all bits set to 0.

File descriptor masks of type **fd_set** can be initialized and tested with *FD_CLR*( ), *FD_ISSET*( ), *FD_SET*( ), and *FD_ZERO*( ). It is unspecified whether each of these is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with any of these names, the behavior is undefined.

FD_CLR(*fd*, *fdsetp*) shall remove the file descriptor *fd* from the set pointed to by *fdsetp*. If *fd* is not a member of this set, there shall be no effect on the set, nor will an error be returned.

FD_ISSET(*fd*, *fdsetp*) shall evaluate to non-zero if the file descriptor *fd* is a member of the set pointed to by *fdsetp*, and shall evaluate to zero otherwise.

FD_SET(*fd*, *fdsetp*) shall add the file descriptor *fd* to the set pointed to by *fdsetp*. If the file descriptor *fd* is already in this set, there shall be no effect on the set, nor will an error be returned.

FD_ZERO(*fdsetp*) shall initialize the descriptor set pointed to by *fdsetp* to the null set. No error is returned if the set is not empty at the time *FD_ZERO*( ) is invoked.

The behavior of these macros is undefined if the *fd* argument is less than 0 or greater than or equal to FD_SETSIZE, or if *fd* is not a valid file descriptor, or if any of the arguments are expressions with side effects.

RETURN VALUE

Upon successful completion, the *pselect*( ) and *select*( ) functions shall return the total number of bits set in the bit masks. Otherwise, −1 shall be returned, and shall set *errno* to indicate the error.

FD_CLR( ), *FD_SET*( ), and *FD_ZERO*( ) not return a value. *FD_ISSET*( ) shall return a non-zero value if the bit for the file descriptor *fd* is set in the file descriptor set pointed to by *fdset*, and 0 otherwise.

ERRORS

Under the following conditions, *pselect*( ) and *select*( ) shall fail and set *errno* to:

[EBADF]	One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.

	[EINTR]	The function was interrupted before any of the selected events occurred and before the timeout interval expired.
XSI		If SA_RESTART has been set for the interrupting signal, it is implementation-defined whether the function restarts or returns with [EINTR].
	[EINVAL]	An invalid timeout interval was specified.
	[EINVAL]	The *nfds* argument is less than 0 or greater than FD_SETSIZE.
XSR	[EINVAL]	One of the specified file descriptors refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

In previous versions of the Single UNIX Specification, the *select*( ) function was defined in the **<sys/time.h>** header. This is now changed to **<sys/select.h>**. The rationale for this change was as follows: the introduction of the *pselect*( ) function included the **<sys/select.h>** header and the **<sys/select.h>** header defines all the related definitions for the *pselect*( ) and *select*( ) functions. Backwards-compatibility to existing XSI implementations is handled by allowing **<sys/time.h>** to include **<sys/select.h>**.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *alarm*( ), *connect*( ), *fcntl*( ), *poll*( ), *read*( ), *recvmsg*( ), *sendmsg*( ), *setitimer*( ), *ualarm*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/select.h>**, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

In the ERRORS section, the text has been changed to indicate that [EINVAL] is returned when *nfds* is less than 0 or greater than FD_SETSIZE. It previously stated less than 0, or greater than or equal to FD_SETSIZE.

Text about *timeout* is moved from the APPLICATION USAGE section to the DESCRIPTION.

Issue 6

The Open Group Corrigendum U026/6 is applied, changing the occurrences of *readfs* and *writefs* in the *select*( ) DESCRIPTION to be *readfds* and *writefds*.

Text referring to sockets is added to the DESCRIPTION.

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- These functions are now mandatory.

The *pselect*( ) function is added for alignment with IEEE Std 1003.1g-2000 and additional detail related to sockets semantics is added to the DESCRIPTION.

The *select*( ) function now requires inclusion of **<sys/select.h>**.

The **restrict** keyword is added to the *select*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_atfork — register fork handlers

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
    void (*child)(void));
```

DESCRIPTION

The *pthread_atfork*( ) function shall declare fork handlers to be called before and after *fork*( ), in the context of the thread that called *fork*( ). The *prepare* fork handler shall be called before *fork*( ) processing commences. The *parent* fork handle shall be called after *fork*( ) processing completes in the parent process. The *child* fork handler shall be called after *fork*( ) processing completes in the child process. If no handling is desired at one or more of these three points, the corresponding fork handler address(es) may be set to NULL.

The order of calls to *pthread_atfork*( ) is significant. The *parent* and *child* fork handlers shall be called in the order in which they were established by calls to *pthread_atfork*( ). The *prepare* fork handlers shall be called in the opposite order.

RETURN VALUE

Upon successful completion, *pthread_atfork*( ) shall return a value of zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_atfork*( ) function shall fail if:

[ENOMEM]	Insufficient table space exists to record the fork handler addresses.

The *pthread_atfork*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

There are at least two serious problems with the semantics of *fork*( ) in a multi-threaded program. One problem has to do with state (for example, memory) covered by mutexes. Consider the case where one thread has a mutex locked and the state covered by that mutex is inconsistent while another thread calls *fork*( ). In the child, the mutex is in the locked state (locked by a nonexistent thread and thus can never be unlocked). Having the child simply reinitialize the mutex is unsatisfactory since this approach does not resolve the question about how to correct or otherwise deal with the inconsistent state in the child.

It is suggested that programs that use *fork*( ) call an *exec* function very soon afterwards in the child process, thus resetting all states. In the meantime, only a short list of async-signal-safe library routines are promised to be available.

Unfortunately, this solution does not address the needs of multi-threaded libraries. Application programs may not be aware that a multi-threaded library is in use, and they feel free to call any number of library routines between the *fork*( ) and *exec* calls, just as they always have. Indeed, they may be extant single-threaded programs and cannot, therefore, be expected to obey new restrictions imposed by the threads library.

On the other hand, the multi-threaded library needs a way to protect its internal state during *fork*( ) in case it is re-entered later in the child process. The problem arises especially in multi-threaded I/O libraries, which are almost sure to be invoked between the *fork*( ) and *exec* calls to effect I/O redirection. The solution may require locking mutex variables during *fork*( ), or it may entail simply resetting the state in the child after the *fork*( ) processing completes.

The *pthread_atfork*( ) function provides multi-threaded libraries with a means to protect themselves from innocent application programs that call *fork*( ), and it provides multi-threaded application programs with a standard mechanism for protecting themselves from *fork*( ) calls in a library routine or the application itself.

The expected usage is that the *prepare* handler acquires all mutex locks and the other two fork handlers release them.

For example, an application can supply a *prepare* routine that acquires the necessary mutexes the library maintains and supply *child* and *parent* routines that release those mutexes, thus ensuring that the child gets a consistent snapshot of the state of the library (and that no mutexes are left stranded). Alternatively, some libraries might be able to supply just a *child* routine that reinitializes the mutexes in the library and all associated states to some known value (for example, what it was when the image was originally executed).

When *fork*( ) is called, only the calling thread is duplicated in the child process. Synchronization variables remain in the same state in the child as they were in the parent at the time *fork*( ) was called. Thus, for example, mutex locks may be held by threads that no longer exist in the child process, and any associated states may be inconsistent. The parent process may avoid this by explicit code that acquires and releases locks critical to the child via *pthread_atfork*( ). In addition, any critical threads need to be recreated and reinitialized to the proper state in the child (also via *pthread_atfork*( )).

A higher-level package may acquire locks on its own data structures before invoking lower-level packages. Under this scenario, the order specified for fork handler calls allows a simple rule of initialization for avoiding package deadlock: a package initializes all packages on which it depends before it calls the *pthread_atfork*( ) function for itself.

FUTURE DIRECTIONS

None.

SEE ALSO

atexit( ), *fork*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 5. Derived from the POSIX Threads Extension.

IEEE PASC Interpretation 1003.1c #4 is applied.

Issue 6

The *pthread_atfork*( ) function is marked as part of the Threads option.

The **<pthread.h>** header is added to the SYNOPSIS.

NAME

pthread_attr_destroy, pthread_attr_init — destroy and initialize threads attributes object

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);

```

DESCRIPTION

The *pthread_attr_destroy*( ) function shall destroy a thread attributes object. An implementation may cause *pthread_attr_destroy*( ) to set *attr* to an implementation-defined invalid value. A destroyed *attr* attributes object can be reinitialized using *pthread_attr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

The *pthread_attr_init*( ) function shall initialize a thread attributes object *attr* with the default value for all of the individual attributes used by a given implementation.

The resulting attributes object (possibly modified by setting individual attribute values), when used by *pthread_create*( ) defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to *pthread_create*( ). Results are undefined if *pthread_attr_init*( ) is called specifying an already initialized *attr* attributes object.

RETURN VALUE

Upon successful completion, *pthread_attr_destroy*( ) and *pthread_attr_init*( ) shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the thread attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the function itself be changed.

Attributes objects provide clean isolation of the configurable aspects of threads. For example, ‘‘stack size’’ is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up ‘‘classes’ of threads with similar attributes; for example, ‘‘threads with large stacks and high priority’’ or ‘‘threads with minimal stacks’’. These classes can be defined in a single place and then referenced wherever threads need to be created. Changes to ‘‘class’’ decisions become straightforward, and detailed analysis of each *pthread_create*( ) call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Since assignment is not necessarily defined on a given opaque type, implementation-defined default values cannot be defined in a portable way. The solution to this problem is to allow attributes objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

The following proposal was provided as a suggested alternative to the supplied attributes:

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (*pthread_create*( ), *pthread_mutex_init*( ), *pthread_cond_init*( )). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.
2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the **pthread_mutex_t** and **pthread_cond_t** objects (instead of on attributes objects).

The difficulties with this solution are:

1. A bitmask is not opaque if bits have to be set into bitvector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, get and set functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.
2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3-bit or 4-bit, but priority requires 5-bit or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by get and set functions (making the bitmask a redundant addition to the attributes objects).

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in this volume of IEEE Std 1003.1-200x has to be done with care so as not to affect binary-compatibility.

Attributes objects, even if allocated by means of dynamic allocation functions such as *malloc*( ), may have their size fixed at compile time. This means, for example, a *pthread_create*( ) in an implementation with extensions to the **pthread_attr_t** cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_getstackaddr( ), *pthread_attr_getstacksize*( ), *pthread_attr_getdetachstate*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_destroy*( ) and *pthread_attr_init*( ) functions marked as part of the Threads option.

IEEE PASC Interpretation 1003.1 #107 is applied, noting that the effect of initializing an already initialized thread attributes object is undefined.

NAME

pthread_attr_getdetachstate, pthread_attr_setdetachstate — get and set detachstate attribute

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
    int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

DESCRIPTION

The *detachstate* attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the *pthread_detach*( ) or *pthread_join*( ) function is an error.

The *pthread_attr_getdetachstate*( ) and *pthread_attr_setdetachstate*( ) functions, respectively, shall get and set the *detachstate* attribute in the *attr* object.

For *pthread_attr_getdetachstate*( ), *detachstate* shall be set to either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

For *pthread_attr_setdetachstate*( ), the application shall set *detachstate* to either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

A value of PTHREAD_CREATE_DETACHED shall cause all threads created with *attr* to be in the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE shall cause all threads created with *attr* to be in the joinable state. The default value of the *detachstate* attribute shall be PTHREAD_CREATE_JOINABLE.

RETURN VALUE

Upon successful completion, *pthread_attr_getdetachstate*( ) and *pthread_attr_setdetachstate*( ) shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The *pthread_attr_getdetachstate*( ) function stores the value of the *detachstate* attribute in *detachstate* if successful.

ERRORS

The *pthread_attr_setdetachstate*( ) function shall fail if:

[EINVAL] The value of *detachstate* was not valid

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getstacksize*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_setdetachstate*( ) and *pthread_attr_getdetachstate*( ) functions are marked as part of the Threads option.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

pthread_attr_getguardsize, pthread_attr_setguardsize — get and set the thread guardsize attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
    size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

DESCRIPTION

The *pthread_attr_getguardsize*( ) function shall get the *guardsize* attribute in the *attr* object. This attribute shall be returned in the *guardsize* parameter.

The *pthread_attr_setguardsize*( ) function shall set the *guardsize* attribute in the *attr* object. The new value of this attribute shall be obtained from the *guardsize* parameter. If *guardsize* is zero, a guard area shall not be provided for threads created with *attr*. If *guardsize* is greater than zero, a guard area of at least size *guardsize* bytes shall be provided for each thread created with *attr*.

The *guardsize* attribute controls the size of the guard area for the created thread's stack. The *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error shall result (possibly in a SIGSEGV signal being delivered to the thread).

A conforming implementation may round up the value contained in *guardsize* to a multiple of the configurable system variable {PAGESIZE} (see **<sys/mman.h>**). If an implementation rounds up the value of *guardsize* to a multiple of {PAGESIZE}, a call to *pthread_attr_getguardsize*( ) specifying *attr* shall store in the *guardsize* parameter the guard size specified by the previous *pthread_attr_setguardsize*( ) function call.

The default value of the *guardsize* attribute is {PAGESIZE} bytes. The actual value of {PAGESIZE} is implementation-defined.

If the *stackaddr* or *stack* attribute has been set (that is, the caller is allocating and managing its own thread stacks), the *guardsize* attribute shall be ignored and no protection shall be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

RETURN VALUE

If successful, the *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_getguardsize*( ) and *pthread_attr_setguardsize*( ) functions shall fail if:

[EINVAL]	The attribute *attr* is invalid.
[EINVAL]	The parameter *guardsize* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *guardsize* attribute is provided to the application for two reasons:

1. Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads never overflow their stack, can save system resources by turning off guard areas.
2. When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

In the ERRORS section, a third [EINVAL] error condition is removed as it is covered by the second error condition.

The **restrict** keyword is added to the *pthread_attr_getguardsize*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_attr_getinheritsched, pthread_attr_setinheritsched — get and set inheritsched attribute (**REALTIME THREADS**)

SYNOPSIS

THR TPS

```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
    int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```

DESCRIPTION

The *pthread_attr_getinheritsched*( ), and *pthread_attr_setinheritsched*( ) functions, respectively, shall get and set the *inheritsched* attribute in the *attr* argument.

When the attributes objects are used by *pthread_create*( ), the *inheritsched* attribute determines how the other scheduling attributes of the created thread shall be set.

PTHREAD_INHERIT_SCHED
: Specifies that the thread scheduling attributes shall be inherited from the creating thread, and the scheduling attributes in this *attr* argument shall be ignored.

PTHREAD_EXPLICIT_SCHED
: Specifies that the thread scheduling attributes shall be set to the corresponding values from this attributes object.

The symbols PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED are defined in the **<pthread.h>** header.

The following ‘‘thread scheduling attributes’’ defined by IEEE Std 1003.1-200x are affected by the *inheritsched* attribute: scheduling policy (*schedpolicy*), scheduling parameters (*schedparam*), and scheduling contention scope (*contentionscope*).

RETURN VALUE

If successful, the *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_setinheritsched*( ) function may fail if:

[EINVAL]	The value of *inheritsched* is not valid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using *pthread_create*( ). Using these routines does not affect the current running thread.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getscope*( ), *pthread_attr_getschedpolicy*( ), *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getinheritsched*( ) and *pthread_attr_setinheritsched*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The **restrict** keyword is added to the *pthread_attr_getinheritsched*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

pthread_attr_getschedparam, pthread_attr_setschedparam — get and set schedparam attribute

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
    struct sched_param *restrict param);
int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

DESCRIPTION

The *pthread_attr_getschedparam*( ), and *pthread_attr_setschedparam*( ) functions, respectively, shall get and set the scheduling parameter attributes in the *attr* argument. The contents of the *param* structure are defined in the **<sched.h>** header. For the SCHED_FIFO and SCHED_RR policies, the only required member of *param* is *sched_priority*. |

TSP For the SCHED_SPORADIC policy, the required members of the *param* structure are *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period* must be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

RETURN VALUE

If successful, the *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_setschedparam*( ) function may fail if:

[EINVAL]	The value of *param* is not valid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using *pthread_create*( ). Using these routines does not affect the current running thread.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getscope*( ), *pthread_attr_getinheritsched*( ), *pthread_attr_getschedpolicy*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) functions are marked as part of the Threads option.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_attr_getschedparam*( ) and *pthread_attr_setschedparam*( ) prototypes for alignment with the ISO/IEC 9899: 1999 standard.

NAME

pthread_attr_getschedpolicy, pthread_attr_setschedpolicy — get and set schedpolicy attribute (**REALTIME THREADS**)

SYNOPSIS

THR TPS
```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
    int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

```

DESCRIPTION

The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions, respectively, shall get and set the *schedpolicy* attribute in the *attr* argument.

The supported values of *policy* shall include SCHED_FIFO, SCHED_RR, and SCHED_OTHER, which are defined in the **<sched.h>** header. When threads executing with the scheduling policy SCHED_FIFO, SCHED_RR, TSP or SCHED_SPORADIC are waiting on a mutex, they shall acquire the mutex in priority order when the mutex is unlocked. |

RETURN VALUE

If successful, the *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_setschedpolicy*( ) function may fail if:

[EINVAL]	The value of *policy* is not valid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using *pthread_create*( ). Using these routines does not affect the current running thread.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getscope*( ), *pthread_attr_getinheritsched*( ), *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getschedpolicy*( ) and *pthread_attr_setschedpolicy*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_attr_getschedpolicy*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_attr_getscope, pthread_attr_setscope — get and set contentionscope attribute (**REALTIME THREADS**)

SYNOPSIS

THR TPS

```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
    int *restrict contentionscope);
int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

DESCRIPTION

The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions, respectively, shall get and set the *contentionscope* attribute in the *attr* object.

The *contentionscope* attribute may have the values PTHREAD_SCOPE_SYSTEM, signifying system scheduling contention scope, or PTHREAD_SCOPE_PROCESS, signifying process scheduling contention scope. The symbols PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS are defined in the **<pthread.h>** header.

RETURN VALUE

If successful, the *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_attr_setscope*( ) function may fail if:

[EINVAL]	The value of *contentionscope* is not valid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

After these attributes have been set, a thread can be created with the specified attributes using *pthread_create*( ). Using these routines does not affect the current running thread.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getinheritsched*( ), *pthread_attr_getschedpolicy*( ), *pthread_attr_getschedparam*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_attr_getscope*( ) and *pthread_attr_setscope*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The **restrict** keyword is added to the *pthread_attr_getscope*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

pthread_attr_getstack, pthread_attr_setstack — get and set stack attributes

SYNOPSIS

THR

```
#include <pthread.h>
```

TSA TSS

```
int pthread_attr_getstack(const pthread_attr_t *restrict attr,
    void **restrict stackaddr, size_t *restrict stacksize);
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

DESCRIPTION

The *pthread_attr_getstack*( ) and *pthread_attr_setstack*( ) functions, respectively, shall get and set the thread creation stack attributes *stackaddr* and *stacksize* in the *attr* object.

The stack attributes specify the area of storage to be used for the created thread's stack. The base (lowest addressable byte) of the storage shall be *stackaddr*, and the size of the storage shall be *stacksize* bytes. The *stacksize* shall be at least {PTHREAD_STACK_MIN}. The *stackaddr* shall be aligned appropriately to be used as a stack; for example, *pthread_attr_setstack*( ) may fail with [EINVAL] if (*stackaddr* & 0x7) is not 0. All pages within the stack described by *stackaddr* and *stacksize* shall be both readable and writable by the thread.

RETURN VALUE

Upon successful completion, these functions shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The *pthread_attr_getstack*( ) function shall store the stack attribute values in *stackaddr* and *stacksize* if successful.

ERRORS

The *pthread_attr_setstack*( ) function shall fail if:

[EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds an implementation-defined limit.

The *pthread_attr_setstack*( ) function may fail if:

[EINVAL] The value of *stackaddr* does not have proper alignment to be used as a stack, or if (*stackaddr* + *stacksize*) lacks proper alignment.

[EACCES] The stack page(s) described by *stackaddr* and *stacksize* are not both readable and writable by the thread.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

These functions are appropriate for use by applications in an environment where the stack for a thread must be placed in some particular region of memory.

While it might seem that an application could detect stack overflow by providing a protected page outside the specified stack region, this cannot be done portably. Implementations are free to place the thread's initial stack pointer anywhere within the specified region to accommodate the machine's stack pointer behavior and allocation requirements. Furthermore, on some architectures, such as the IA-64, ''overflow'' might mean that two separate stack pointers allocated within the region will overlap somewhere in the middle of the region.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_init( ), *pthread_attr_setdetachstate*( ), *pthread_attr_setstacksize*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Developed as an XSI extension and brought into the BASE by IEEE PASC Interpretation 1003.1 #101.

NAME

pthread_attr_getstackaddr, pthread_attr_setstackaddr — get and set stackaddr attribute

SYNOPSIS

THR TSA

```
#include <pthread.h>
```

OB

```
int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
    void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

DESCRIPTION

The *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) functions, respectively, shall get and set the thread creation *stackaddr* attribute in the *attr* object.

The *stackaddr* attribute specifies the location of storage to be used for the created thread's stack. The size of the storage shall be at least {PTHREAD_STACK_MIN}.

RETURN VALUE

Upon successful completion, *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The *pthread_attr_getstackaddr*( ) function stores the *stackaddr* attribute value in *stackaddr* if successful.

ERRORS

No errors are defined.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The specification of the *stackaddr* attribute presents several ambiguities that make portable use of these interfaces impossible. The description of the single address parameter as a ''stack'' does not specify a particular relationship between the address and the ''stack'' implied by that address. For example, the address may be taken as the low memory address of a buffer intended for use as a stack, or it may be taken as the address to be used as the initial stack pointer register value for the new thread. These two are not the same except for a machine on which the stack grows ''up'' from low memory to high, and on which a ''push'' operation first stores the value in memory and then increments the stack pointer register. Further, on a machine where the stack grows ''down'' from high memory to low, interpretation of the address as the ''low memory'' address requires a determination of the intended size of the stack. IEEE Std 1003.1-200x has introduced the new interfaces *pthread_attr_setstack*( ) and *pthread_attr_getstack*( ) to resolve these ambiguities.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getdetachstate*( ), *pthread_attr_getstack*( ), *pthread_attr_getstacksize*( ), *pthread_attr_setstack*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_getstackaddr*( ) and *pthread_attr_setstackaddr*( ) functions are marked as part of the Threads and Thread Stack Address Attribute options.

The **restrict** keyword is added to the *pthread_attr_getstackaddr*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

These functions are marked obsolescent.

NAME

pthread_attr_getstacksize, pthread_attr_setstacksize — get and set stacksize attribute

SYNOPSIS

THR TSA

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
    size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

DESCRIPTION

The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions, respectively, shall get and set the thread creation *stacksize* attribute in the *attr* object.

The *stacksize* attribute shall define the minimum stack size (in bytes) allocated for the created threads stack.

RETURN VALUE

Upon successful completion, *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) shall return a value of 0; otherwise, an error number shall be returned to indicate the error.

The *pthread_attr_getstacksize*( ) function stores the *stacksize* attribute value in *stacksize* if successful.

ERRORS

The *pthread_attr_setstacksize*( ) function shall fail if:

[EINVAL] The value of *stacksize* is less than {PTHREAD_STACK_MIN} or exceeds a system-imposed limit.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_attr_destroy( ), *pthread_attr_getstackaddr*( ), *pthread_attr_getdetachstate*( ), *pthread_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_attr_getstacksize*( ) and *pthread_attr_setstacksize*( ) functions are marked as part of the Threads and Thread Stack Address Attribute options.

The **restrict** keyword is added to the *pthread_attr_getstacksize*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_attr_init — initialize threads attributes object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

DESCRIPTION

Refer to *pthread_attr_destroy*( ).

NAME

pthread_attr_setdetachstate — set detachstate attribute

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

DESCRIPTION

Refer to *pthread_attr_getdetachstate*( ).

NAME

pthread_attr_setguardsize — set thread guardsize attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr,
    size_t guardsize);
```

DESCRIPTION

Refer to *pthread_attr_getguardsize*( ).

NAME

pthread_attr_setinheritsched — set inheritsched attribute (**REALTIME THREADS**)

SYNOPSIS

THR TPS
```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t *attr,
    int inheritsched);
```

DESCRIPTION

Refer to *pthread_attr_getinheritsched*( ).

NAME

pthread_attr_setschedparam — set schedparam attribute

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
    const struct sched_param *restrict param);
```

DESCRIPTION

Refer to *pthread_attr_getschedparam*( ).

NAME

pthread_attr_setschedpolicy — set schedpolicy attribute (**REALTIME THREADS**)

SYNOPSIS

THR TPS

```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

DESCRIPTION

Refer to *pthread_attr_getschedpolicy*( ).

NAME

pthread_attr_setscope — set contentionscope attribute (**REALTIME THREADS**)

SYNOPSIS

THR TPS

```
#include <pthread.h>

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

DESCRIPTION

Refer to *pthread_attr_getscope*( ).

NAME

pthread_attr_setstack — set stack attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
    size_t stacksize);
```

DESCRIPTION

Refer to *pthread_attr_getstack*( ).

NAME

pthread_attr_setstackaddr — set stackaddr attribute

SYNOPSIS

THR TSA

```
#include <pthread.h>
```

OB

```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

DESCRIPTION

Refer to *pthread_attr_getstackaddr*( ).

NAME

pthread_attr_setstacksize — set stacksize attribute

SYNOPSIS

THR TSA

```
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

DESCRIPTION

Refer to *pthread_attr_getstacksize*( ).

NAME

pthread_barrier_destroy, pthread_barrier_init — destroy and initialize a barrier object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR BAR
```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
```

DESCRIPTION

The *pthread_barrier_destroy*( ) function shall destroy the barrier referenced by *barrier* and release any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to *pthread_barrier_init*( ). An implementation may use this function to set *barrier* to an invalid value. The results are undefined if *pthread_barrier_destroy*( ) is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

The *pthread_barrier_init*( ) function shall allocate any resources required to use the barrier referenced by *barrier* and shall initialize the barrier with attributes referenced by *attr*. If *attr* is NULL, the default barrier attributes shall be used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if *pthread_barrier_init*( ) is called when any thread is blocked on the barrier (that is, has not returned from the *pthread_barrier_wait*( ) call). The results are undefined if a barrier is used without first being initialized. The results are undefined if *pthread_barrier_init*( ) is called specifying an already initialized barrier.

The *count* argument specifies the number of threads that must call *pthread_barrier_wait*( ) before any of them successfully return from the call. The value specified by *count* must be greater than zero.

If the *pthread_barrier_init*( ) function fails, the barrier shall not be initialized and the contents of *barrier* are undefined. |

Only the object referenced by *barrier* may be used for performing synchronization. The result of referring to copies of that object in calls to *pthread_barrier_destroy*( ) or *pthread_barrier_wait*( ) is undefined.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_barrier_destroy*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*( ) call) by another thread.
[EINVAL]	The value specified by *barrier* is invalid.

The *pthread_barrier_init*( ) function shall fail if:

[EAGAIN]	The system lacks the necessary resources to initialize another barrier.
[EINVAL]	The value specified by *count* is equal to zero.

[ENOMEM]	Insufficient memory exists to initialize the barrier.

The *pthread_barrier_init*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a *pthread_barrier_wait*( ) call) by another thread.
[EINVAL]	The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_barrier_destroy*( ) and *pthread_barrier_init*( ) functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_wait( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

pthread_barrier_init — initialize a barrier object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR BAR
```
#include <pthread.h>

int pthread_barrier_init(pthread_barrier_t *restrict barrier,
    const pthread_barrierattr_t *restrict attr, unsigned count);
```

DESCRIPTION

Refer to *pthread_barrier_destroy*( ).

NAME

pthread_barrier_wait — synchronize at a barrier (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR BAR
```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

DESCRIPTION

The *pthread_barrier_wait*( ) function shall synchronize participating threads at the barrier referenced by *barrier*. The calling thread shall block until the required number of threads have called *pthread_barrier_wait*( ) specifying the barrier.

When the required number of threads have called *pthread_barrier_wait*( ) specifying the barrier, the constant PTHREAD_BARRIER_SERIAL_THREAD shall be returned to one unspecified thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall be reset to the state it had as a result of the most recent *pthread_barrier_init*( ) function that referenced it.

The constant PTHREAD_BARRIER_SERIAL_THREAD is defined in **<pthread.h>** and its value shall be distinct from any other value returned by *pthread_barrier_wait*( ).

The results are undefined if this function is called with an uninitialized barrier.

If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler the thread shall resume waiting at the barrier if the barrier wait has not completed (that is, if the required number of threads have not arrived at the barrier during the execution of the signal handler); otherwise, the thread shall continue as normal from the completed barrier wait. Until the thread in the signal handler returns from it, it is unspecified whether other threads may proceed past the barrier once they have all reached it.

A thread that has blocked on a barrier shall not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources shall be determined by the scheduling policy.

RETURN VALUE

Upon successful completion, the *pthread_barrier_wait*( ) function shall return PTHREAD_BARRIER_SERIAL_THREAD for a single (arbitrary) thread synchronized at the barrier and zero for each of the other threads. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_barrier_wait*( ) function may fail if:

[EINVAL] The value specified by *barrier* does not refer to an initialized barrier object.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *pthread_barrier_wait*( ) function is part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_destroy( ), *pthread_barrier_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_barrierattr_destroy, pthread_barrierattr_init — destroy and initialize barrier attributes object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR BAR

```
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

DESCRIPTION

The *pthread_barrierattr_destroy*( ) function shall destroy a barrier attributes object. A destroyed *attr* attributes object can be reinitialized using *pthread_barrierattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause *pthread_barrierattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

The *pthread_barrierattr_init*( ) function shall initialize a barrier attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if *pthread_barrierattr_init*( ) is called specifying an already initialized *attr* attributes object.

After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) shall not affect any previously initialized barrier.

RETURN VALUE

If successful, the *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_barrierattr_destroy*( ) function may fail if:

[EINVAL] The value specified by *attr* is invalid.

The *pthread_barrierattr_init*( ) function shall fail if:

[ENOMEM] Insufficient memory exists to initialize the barrier attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_barrierattr_destroy*( ) and *pthread_barrierattr_init*( ) functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrierattr_getpshared( ), *pthread_barrierattr_setpshared*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**.

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_barrierattr_getpshared, pthread_barrierattr_setpshared — get and set process-shared attribute of barrier attributes object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR
```
#include <pthread.h>
```

BAR TSH
```
int pthread_barrierattr_getpshared(
    const pthread_barrierattr_t *restrict attr,
    int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```

DESCRIPTION

The *pthread_barrierattr_getpshared*( ) function shall obtain the value of the process-shared attribute from the attributes object referenced by *attr*. The *pthread_barrierattr_setpshared*( ) function shall set the process-shared attribute in an initialized attributes object referenced by *attr*.

The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the barrier shall only be operated upon by threads created within the same process as the thread that initialized the barrier; if threads of different processes attempt to operate on such a barrier, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE. Both constants PTHREAD_PROCESS_SHARED and PTHREAD_PROCESS_PRIVATE are defined in **<pthread.h>**.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

RETURN VALUE

If successful, the *pthread_barrierattr_getpshared*( ) function shall return zero and store the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_barrierattr_setpshared*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_barrierattr_setpshared*( ) function may fail if:

[EINVAL]	The new value specified for the process-shared attribute is not one of the legal values PTHREAD_PROCESS_SHARED or PTHREAD_PROCESS_PRIVATE.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_barrierattr_getpshared*( ) and *pthread_barrierattr_setpshared*( ) functions are part of the Barriers option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_barrier_init( ), *pthread_barrierattr_destroy*( ), *pthread_barrierattr_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000

NAME

pthread_barrierattr_init — initialize barrier attributes object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR BAR

```
#include <pthread.h>

int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

DESCRIPTION

Refer to *pthread_barrierattr_destroy*( ).

NAME

pthread_barrierattr_setpshared — set process-shared attribute of barrier attributes object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR
```
#include <pthread.h>
```

BAR TSH
```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
    int pshared);
```

DESCRIPTION

Refer to *pthread_barrierattr_getpshared*( ).

NAME

pthread_cancel — cancel execution of a thread

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

DESCRIPTION

The *pthread_cancel*( ) function shall request that *thread* be canceled. The target thread's cancelability state and type determines when the cancelation takes effect. When the cancelation is acted on, the cancelation cleanup handlers for *thread* shall be called. When the last cancelation cleanup handler returns, the thread-specific data destructor functions shall be called for *thread*. When the last destructor function returns, *thread* shall be terminated.

The cancelation processing in the target thread shall run asynchronously with respect to the calling thread returning from *pthread_cancel*( ).

RETURN VALUE

If successful, the *pthread_cancel*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cancel*( ) function may fail if:

[ESRCH] No thread could be found corresponding to that specified by the given thread ID.

The *pthread_cancel*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Two alternative functions were considered to sending the cancelation notification to a thread. One would be to define a new SIGCANCEL signal that had the cancelation semantics when delivered; the other was to define the new *pthread_cancel*( ) function, which would trigger the cancelation semantics.

The advantage of a new signal was that so much of the delivery criteria were identical to that used when trying to deliver a signal that making cancelation notification a signal was seen as consistent. Indeed, many implementations implement cancelation using a special signal. On the other hand, there would be no signal functions that could be used with this signal except *pthread_kill*( ), and the behavior of the delivered cancelation signal would be unlike any previously existing defined signal.

The benefits of a special function include the recognition that this signal would be defined because of the similar delivery criteria and that this is the only common behavior between a cancelation request and a signal. In addition, the cancelation delivery mechanism does not have to be implemented as a signal. There are also strong, if not stronger, parallels with language exception mechanisms than with signals that are potentially obscured if the delivery mechanism is visibly closer to signals.

In the end, it was considered that as there were so many exceptions to the use of the new signal with existing signals functions that it would be misleading. A special function has resolved this problem. This function was carefully defined so that an implementation wishing to provide the cancelation functions on top of signals could do so. The special function also means that implementations are not obliged to implement cancelation with signals.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_exit( ), *pthread_cond_wait*( ), *pthread_cond_timedwait*( ), *pthread_join*( ), *pthread_setcancelstate*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cancel*( ) function is marked as part of the Threads option.

NAME

pthread_cleanup_pop, pthread_cleanup_push — establish cancelation handlers

SYNOPSIS

THR

```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

DESCRIPTION

The *pthread_cleanup_pop*( ) function shall remove the routine at the top of the calling thread's cancelation cleanup stack and optionally invoke it (if *execute* is non-zero).

The *pthread_cleanup_push*( ) function shall push the specified cancelation cleanup handler *routine* onto the calling thread's cancelation cleanup stack. The cancelation cleanup handler shall be popped from the cancelation cleanup stack and invoked with the argument *arg* when:

- The thread exits (that is, calls *pthread_exit*( )).
- The thread acts upon a cancelation request.
- The thread calls *pthread_cleanup_pop*( ) with a non-zero *execute* argument.

These functions may be implemented as macros. The application shall ensure that they appear as statements, and in pairs within the same lexical scope (that is, the *pthread_cleanup_push*( ) macro may be thought to expand to a token list whose first token is `'{'` with *pthread_cleanup_pop*( ) expanding to a token list whose last token is the corresponding `'}'`).

The effect of calling *longjmp*( ) or *siglongjmp*( ) is undefined if there have been any calls to *pthread_cleanup_push*( ) or *pthread_cleanup_pop*( ) made without the matching call since the jump buffer was filled. The effect of calling *longjmp*( ) or *siglongjmp*( ) from inside a cancelation cleanup handler is also undefined unless the jump buffer was also filled in the cancelation cleanup handler.

RETURN VALUE

The *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ) functions shall not return a value.

ERRORS

No errors are defined.

These functions shall not return an error code of [EINTR].

EXAMPLES

The following is an example using thread primitives to implement a cancelable, writers-priority read-write lock:

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
        wcond;
    int lock_count; /* < 0 .. Held by writer. */
                    /* > 0 .. Held by lock_count readers. */
                    /* = 0 .. Held by nobody. */
    int waiting_writers; /* Count of waiting writers. */
} rwlock;

void
waiting_reader_cleanup(void *arg)
{
```

```
    rwlock *l;

    l = (rwlock *) arg;
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_read(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
    while ((l->lock_count < 0) && (l->waiting_writers != 0))
        pthread_cond_wait(&l->rcond, &l->lock);
    l->lock_count++;
   /*
    * Note the pthread_cleanup_pop executes
    * waiting_reader_cleanup.
    */
    pthread_cleanup_pop(1);
}

void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(l);
}

void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
       /*
        * This only happens if we have been canceled.
        */
        pthread_cond_broadcast(&l->wcond);
}
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = −1;
   /*
```

```
         * Note the pthread_cleanup_pop executes
         * waiting_writer_cleanup.
         */
        pthread_cleanup_pop(1);
    }
    void
    release_write_lock(rwlock *l)
    {
        pthread_mutex_lock(&l->lock);
        l->lock_count = 0;
        if (l->waiting_writers == 0)
            pthread_cond_broadcast(&l->rcond)
        else
            pthread_cond_signal(&l->wcond);
        pthread_mutex_unlock(&l->lock);
    }
    /*
     * This function is called to initialize the read/write lock.
     */
    void
    initialize_rwlock(rwlock *l)
    {
        pthread_mutex_init(&l->lock, pthread_mutexattr_default);
        pthread_cond_init(&l->wcond, pthread_condattr_default);
        pthread_cond_init(&l->rcond, pthread_condattr_default);
        l->lock_count = 0;
        l->waiting_writers = 0;
    }
    reader_thread()
    {
        lock_for_read(&lock);
        pthread_cleanup_push(release_read_lock, &lock);
       /*
        * Thread has read lock.
        */
        pthread_cleanup_pop(1);
    }
    writer_thread()
    {
        lock_for_write(&lock);
        pthread_cleanup_push(release_write_lock, &lock);
       /*
        * Thread has write lock.
        */
    pthread_cleanup_pop(1);
    }
```

APPLICATION USAGE

The two routines that push and pop cancelation cleanup handlers, *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ), can be thought of as left and right parentheses. They always need to be matched.

RATIONALE

The restriction that the two routines that push and pop cancelation cleanup handlers, *pthread_cleanup_push*( ) and *pthread_cleanup_pop*( ), have to appear in the same lexical scope allows for efficient macro or compiler implementations and efficient storage management. A sample implementation of these routines as macros might look like this:

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, **__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler;

#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

A more ambitious implementation of these routines might do even better by allowing the compiler to note that the cancelation cleanup handler is a constant and can be expanded inline.

This volume of IEEE Std 1003.1-200x currently leaves unspecified the effect of calling *longjmp*( ) from a signal handler executing in a POSIX System Interfaces function. If an implementation wants to allow this and give the programmer reasonable behavior, the *longjmp*( ) function has to call all cancelation cleanup handlers that have been pushed but not popped since the time *setjmp*( ) was called.

Consider a multi-threaded function called by a thread that uses signals. If a signal were delivered to a signal handler during the operation of *qsort*( ) and that handler were to call *longjmp*( ) (which, in turn, did *not* call the cancelation cleanup handlers) the helper threads created by the *qsort*( ) function would not be canceled. Instead, they would continue to execute and write into the argument array even though the array might have been popped off of the stack.

Note that the specified cleanup handling mechanism is especially tied to the C language and, while the requirement for a uniform mechanism for expressing cleanup is language-independent, the mechanism used in other languages may be quite different. In addition, this mechanism is really only necessary due to the lack of a real exception mechanism in the C language, which would be the ideal solution.

There is no notion of a cancelation cleanup-safe function. If an application has no cancelation points in its signal handlers, blocks any signal whose handler may have cancelation points while calling async-unsafe functions, or disables cancelation while calling async-unsafe functions, all functions may be safely called from cancelation cleanup routines.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cancel( ), *pthread_setcancelstate*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cleanup_pop*( ) and *pthread_cleanup_push*( ) functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

pthread_cond_broadcast, pthread_cond_signal — broadcast or signal a condition

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);

```

DESCRIPTION

These functions shall unblock threads blocked on a condition variable.

The *pthread_cond_broadcast*( ) function shall unblock all threads currently blocked on the specified condition variable *cond*.

The *pthread_cond_signal*( ) function shall unblock at least one of the threads that are blocked on the specified condition variable *cond* (if any threads are blocked on *cond*).

If more than one thread is blocked on a condition variable, the scheduling policy shall determine the order in which threads are unblocked. When each thread unblocked as a result of a *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) returns from its call to *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ), the thread shall own the mutex with which it called *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ). The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy (if applicable), and as if each had called *pthread_mutex_lock*( ).

The *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) functions may be called by a thread whether or not it currently owns the mutex that threads calling *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex shall be locked by the thread calling *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ).

The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall have no effect if there are no threads currently blocked on *cond*.

RETURN VALUE

If successful, the *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) function may fail if:

[EINVAL] The value *cond* does not refer to an initialized condition variable.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_cond_broadcast*( ) function is used whenever the shared-variable state has been changed in a way that more than one thread can proceed with its task. Consider a single producer/multiple consumer problem, where the producer can insert multiple items on a list that is accessed one item at a time by the consumers. By calling the *pthread_cond_broadcast*( ) function, the producer would notify all consumers that might be waiting, and thereby the application would receive more throughput on a multi-processor. In addition, *pthread_cond_broadcast*( ) makes it easier to implement a read-write lock. The *pthread_cond_broadcast*( ) function is needed in order to wake up all waiting readers when a writer releases its lock. Finally, the two-phase commit algorithm can use this broadcast function to notify all clients of an impending transaction commit.

It is not safe to use the *pthread_cond_signal*( ) function in a signal handler that is invoked asynchronously. Even if it were safe, there would still be a race between the test of the Boolean *pthread_cond_wait*( ) that could not be efficiently eliminated.

Mutexes and condition variables are thus not suitable for releasing a waiting thread by signaling from code running in a signal handler.

RATIONALE

Multiple Awakenings by Condition Signal

On a multi-processor, it may be impossible for an implementation of *pthread_cond_signal*( ) to avoid the unblocking of more than one thread blocked on a condition variable. For example, consider the following partial implementation of *pthread_cond_wait*( ) and *pthread_cond_signal*( ), executed by two threads in the order given. One thread is trying to wait on the condition variable, another is concurrently executing *pthread_cond_signal*( ), while a third thread is already waiting.

```
pthread_cond_wait(mutex, cond):
    value = cond->value; /* 1 */
    pthread_mutex_unlock(mutex); /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) { /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex); /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++; /* 4 */
    if (cond->waiter) { /* 5 */
        sleeper = cond->waiter; /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper); /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

The effect is that more than one thread can return from its call to *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) as a result of one call to *pthread_cond_signal*( ). This effect is called ‘‘spurious wakeup’’. Note that the situation is self-correcting in that the number of threads that are so awakened is finite; for example, the next thread to call *pthread_cond_wait*( ) after the sequence of events above blocks.

While this problem could be resolved, the loss of efficiency for a fringe condition that occurs only rarely is unacceptable, especially given that one has to check the predicate associated with a condition variable anyway. Correcting this problem would unnecessarily reduce the degree of concurrency in this basic building block for all higher-level synchronization operations.

An added benefit of allowing spurious wakeups is that applications are forced to code a predicate-testing-loop around the condition wait. This also makes the application tolerate superfluous condition broadcasts or signals on the same condition variable that may be coded in some other part of the application. The resulting applications are thus more robust. Therefore, IEEE Std 1003.1-200x explicitly documents that spurious wakeups may occur.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cond_broadcast*( ) and *pthread_cond_signal*( ) functions are marked as part of the Threads option.

The APPLICATION USAGE section is added.

NAME

pthread_cond_destroy, pthread_cond_init — destroy and initialize condition variables

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

The *pthread_cond_destroy*( ) function shall destroy the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause *pthread_cond_destroy*( ) to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be reinitialized using *pthread_cond_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

The *pthread_cond_init*( ) function shall initialize the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes shall be used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable shall become initialized.

Only *cond* itself may be used for performing synchronization. The result of referring to copies of *cond* in calls to *pthread_cond_wait*( ), *pthread_cond_timedwait*( ), *pthread_cond_signal*( ), *pthread_cond_broadcast*( ), and *pthread_cond_destroy*( ) is undefined.

Attempting to initialize an already initialized condition variable results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_cond_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, shall act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

ERRORS

The *pthread_cond_destroy*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (for example, while being used in a *pthread_cond_wait*( ) or *pthread_cond_timedwait*( )) by another thread.
[EINVAL]	The value specified by *cond* is invalid.

The *pthread_cond_init*( ) function shall fail if:

[EAGAIN]	The system lacked the necessary resources (other than memory) to initialize another condition variable.

[ENOMEM]	Insufficient memory exists to initialize the condition variable.

The *pthread_cond_init*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by *cond*, a previously initialized, but not yet destroyed, condition variable.
[EINVAL]	The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

A condition variable can be destroyed immediately after all the threads that are blocked on it are awakened. For example, consider the following code:

```
struct list {
    pthread_mutex_t lm;
    ...
}
struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
}
/* Find a list element and reserve it. */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;
    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}
delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... remove ep from list ...
    ep->busy = 0;  /* Paranoid. */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&rp->notbusy);
    free(ep);
}
```

In this example, the condition variable and its list element may be freed (line B) immediately after all threads waiting for it are awakened (line A), since the mutex and the code ensure that no other thread can touch the element to be deleted.

APPLICATION USAGE

None.

RATIONALE

See *pthread_mutex_init*( ); a similar rationale applies to condition variables.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_broadcast( ), *pthread_cond_signal*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cond_destroy*( ) and *pthread_cond_init*( ) functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The **restrict** keyword is added to the *pthread_cond_init*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_cond_init — initialize condition variables

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *restrict cond,
    const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

Refer to *pthread_cond_destroy*( ).

NAME

pthread_cond_signal — signal a condition

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

Refer to *pthread_cond_broadcast*( ).

NAME

pthread_cond_timedwait, pthread_cond_wait — wait on a condition

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions shall block on a condition variable. They shall be called with *mutex* locked by the calling thread or undefined behavior results.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means ‘‘atomically with respect to access by another thread to the mutex and then the condition variable’’. That is, if another thread is able to acquire the mutex after the about-to-block thread has released it, then a subsequent call to *pthread_cond_broadcast*( ) or *pthread_cond_signal*( ) in that thread shall behave as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex shall have been locked and shall be owned by the calling thread.

When using condition variables there is always a Boolean predicate involving shared variables associated with each condition wait that is true if the thread should proceed. Spurious wakeups from the *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) functions may occur. Since the return from *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) does not imply anything about the value of this predicate, the predicate should be re-evaluated upon such return.

The effect of using more than one mutex for concurrent *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding shall end when the wait returns.

A condition wait (whether timed or not) is a cancelation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancelation request while in a condition wait is that the mutex is (in effect) re-acquired before calling the first cancelation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), but at that point notices the cancelation request and instead of returning to the caller of *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ), starts the thread cancelation activities, which includes calling cancelation cleanup handlers.

A thread that has been unblocked because it has been canceled while blocked in a call to *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) shall not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The *pthread_cond_timedwait*( ) function shall be equivalent to *pthread_cond_wait*( ), except that an error is returned if the absolute time specified by *abstime* passes (that is, system time equals or exceeds *abstime*) before the condition *cond* is signaled or broadcasted, or if the absolute time

CS

specified by *abstime* has already been passed at the time of the call. If the Clock Selection option is supported, the condition variable shall have a clock attribute which specifies the clock that shall be used to measure the time specified by the *abstime* argument. When such timeouts occur, *pthread_cond_timedwait*( ) shall nonetheless release and re-acquire the mutex referenced by *mutex*. The *pthread_cond_timedwait*( ) function is also a cancelation point.

If a signal is delivered to a thread waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it shall return zero due to spurious wakeup.

RETURN VALUE

Except in the case of [ETIMEDOUT], all these error checks shall act as if they were performed immediately at the beginning of processing for the function and shall cause an error return, in effect, prior to modifying the state of the mutex specified by *mutex* or the condition variable specified by *cond*.

Upon successful completion, a value of zero shall be returned; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_cond_timedwait*( ) function shall fail if:

[ETIMEDOUT]	The time specified by *abstime* to *pthread_cond_timedwait*( ) has passed.

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions may fail if:

[EINVAL]	The value specified by *cond*, *mutex*, or *abstime* is invalid.
[EINVAL]	Different mutexes were supplied for concurrent *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) operations on the same condition variable.
[EPERM]	The mutex was not owned by the current thread at the time of the call.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Condition Wait Semantics

It is important to note that when *pthread_cond_wait*( ) and *pthread_cond_timedwait*( ) return without error, the associated predicate may still be false. Similarly, when *pthread_cond_timedwait*( ) returns with the timeout error, the associated predicate may be true due to an unavoidable race between the expiration of the timeout and the predicate state change.

Some implementations, particularly on a multi-processor, may sometimes cause multiple threads to wake up when the condition variable is signaled simultaneously on different processors.

In general, whenever a condition wait returns, the thread has to re-evaluate the predicate associated with the condition wait to determine whether it can safely proceed, should wait again, or should declare a timeout. A return from the wait does not imply that the associated predicate is either true or false.

It is thus recommended that a condition wait be enclosed in the equivalent of a ‘‘while loop’’ that checks the predicate.

Timed Wait Semantics

An absolute time measure was chosen for specifying the timeout parameter for two reasons. First, a relative time measure can be easily implemented on top of a function that specifies absolute time, but there is a race condition associated with specifying an absolute timeout on top of a function that specifies relative timeouts. For example, assume that *clock_gettime*( ) returns the current time and *cond_relative_timed_wait*( ) uses relative timeouts:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time -now;
cond_relative_timed_wait(c, m, &reltime);
```

If the thread is preempted between the first statement and the last statement, the thread blocks for too long. Blocking, however, is irrelevant if an absolute timeout is used. An absolute timeout also need not be recomputed if it is used multiple times in a loop, such as that enclosing a condition wait.

For cases when the system clock is advanced discontinuously by an operator, it is expected that implementations process any timed wait expiring at an intervening time as if that time had actually occurred.

Cancelation and Condition Wait

A condition wait, whether timed or not, is a cancelation point. That is, the functions *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) are points where a pending (or concurrent) cancelation request is noticed. The reason for this is that an indefinite wait is possible at these points—whatever event is being waited for, even if the program is totally correct, might never occur; for example, some input data being awaited might never be sent. By making condition wait a cancelation point, the thread can be canceled and perform its cancelation cleanup handler even though it may be stuck in some indefinite wait.

A side effect of acting on a cancelation request while a thread is blocked on a condition variable is to re-acquire the mutex before calling any of the cancelation cleanup handlers. This is done in order to ensure that the cancelation cleanup handler is executed in the same state as the critical code that lies both before and after the call to the condition wait function. This rule is also required when interfacing to POSIX threads from languages, such as Ada or C++, which may choose to map cancelation onto a language exception; this rule ensures that each exception handler guarding a critical section can always safely depend upon the fact that the associated mutex has already been locked regardless of exactly where within the critical section the exception was raised. Without this rule, there would not be a uniform rule that exception handlers could follow regarding the lock, and so coding would become very cumbersome.

Therefore, since *some* statement has to be made regarding the state of the lock when a cancelation is delivered during a wait, a definition has been chosen that makes application coding most convenient and error free.

When acting on a cancelation request while a thread is blocked on a condition variable, the implementation is required to ensure that the thread does not consume any condition signals directed at that condition variable if there are any other threads waiting on that condition variable. This rule is specified in order to avoid deadlock conditions that could occur if these two independent requests (one acting on a thread and the other acting on the condition variable) were not processed independently.

Performance of Mutexes and Condition Variables

Mutexes are expected to be locked only for a few instructions. This practice is almost automatically enforced by the desire of programmers to avoid long serial regions of execution (which would reduce total effective parallelism).

When using mutexes and condition variables, one tries to ensure that the usual case is to lock the mutex, access shared data, and unlock the mutex. Waiting on a condition variable should be a relatively rare situation. For example, when implementing a read-write lock, code that acquires a read-lock typically needs only to increment the count of readers (under mutual-exclusion) and return. The calling thread would actually wait on the condition variable only when there is already an active writer. So the efficiency of a synchronization operation is bounded by the cost of mutex lock/unlock and not by condition wait. Note that in the usual case there is no context switch.

This is not to say that the efficiency of condition waiting is unimportant. Since there needs to be at least one context switch per Ada rendezvous, the efficiency of waiting on a condition variable is important. The cost of waiting on a condition variable should be little more than the minimal cost for a context switch plus the time to unlock and lock the mutex.

Features of Mutexes and Condition Variables

It had been suggested that the mutex acquisition and release be decoupled from condition wait. This was rejected because it is the combined nature of the operation that, in fact, facilitates realtime implementations. Those implementations can atomically move a high-priority thread between the condition variable and the mutex in a manner that is transparent to the caller. This can prevent extra context switches and provide more deterministic acquisition of a mutex when the waiting thread is signaled. Thus, fairness and priority issues can be dealt with directly by the scheduling discipline. Furthermore, the current condition wait operation matches existing practice.

Scheduling Behavior of Mutexes and Condition Variables

Synchronization primitives that attempt to interfere with scheduling policy by specifying an ordering rule are considered undesirable. Threads waiting on mutexes and condition variables are selected to proceed in an order dependent upon the scheduling policy rather than in some fixed order (for example, FIFO or priority). Thus, the scheduling policy determines which thread(s) are awakened and allowed to proceed.

Timed Condition Wait

The *pthread_cond_timedwait*( ) function allows an application to give up waiting for a particular condition after a given amount of time. An example of its use follows:

```
(void) pthread_mutex_lock(&t.mn);
        t.waiters++;
    clock_gettime(CLOCK_REALTIME, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_timedwait(&t.cond, &t.mn, &ts);
    t.waiters--;
    if (rc == 0) setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

By making the timeout parameter absolute, it does not need to be recomputed each time the program checks its blocking predicate. If the timeout was relative, it would have to be recomputed before each call. This would be especially difficult since such code would need to take into account the possibility of extra wakeups that result from extra broadcasts or signals on the condition variable that occur before either the predicate is true or the timeout is due.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_signal( ), *pthread_cond_broadcast*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) functions are marked as part of the Threads option.

The Open Group Corrigendum U021/9 is applied, correcting the prototype for the *pthread_cond_wait*( ) function.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for the Clock Selection option.

The ERRORS section has an additional case for [EPERM] in response to IEEE PASC Interpretation 1003.1c #28.

The **restrict** keyword is added to the *pthread_cond_timedwait*( ) and *pthread_cond_wait*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

NAME

pthread_cond_wait — wait on a condition

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
```

DESCRIPTION

Refer to *pthread_cond_timedwait*( ).

NAME

pthread_condattr_destroy, pthread_condattr_init — destroy and initialize condition variable attributes object

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

DESCRIPTION

The *pthread_condattr_destroy*( ) function shall destroy a condition variable attributes object; the object becomes, in effect, uninitialized. An implementation may cause *pthread_condattr_destroy*( ) to set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be reinitialized using *pthread_condattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined. |

The *pthread_condattr_init*( ) function shall initialize a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if *pthread_condattr_init*( ) is called specifying an already initialized *attr* attributes object. |

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) shall not affect any previously initialized condition variables. |

This volume of IEEE Std 1003.1-200x requires two attributes, the *clock* attribute and the *process-shared* attribute.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

RETURN VALUE

If successful, the *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_condattr_destroy*( ) function may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_condattr_init*( ) function shall fail if:

[ENOMEM]	Insufficient memory exists to initialize the condition variable attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See *pthread_attr_init*( ) and *pthread_mutex_init*( ).

A *process-shared* attribute has been defined for condition variables for the same reason it has been defined for mutexes.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_condattr_getpshared*( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_condattr_destroy*( ) and *pthread_condattr_init*( ) functions are marked as part of the Threads option.

NAME

pthread_condattr_getclock, pthread_condattr_setclock — get and set the clock selection condition variable attribute (**ADVANCED REALTIME**)

SYNOPSIS

THR CS

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
    clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

DESCRIPTION

The *pthread_condattr_getclock*( ) function shall obtain the value of the *clock* attribute from the attributes object referenced by *attr*. The *pthread_condattr_setclock*( ) function shall set the *clock* attribute in an initialized attributes object referenced by *attr*. If *pthread_condattr_setclock*( ) is called with a *clock_id* argument that refers to a CPU-time clock, the call shall fail.

The *clock* attribute is the clock ID of the clock that shall be used to measure the timeout service of *pthread_cond_timedwait*( ). The default value of the *clock* attribute shall refer to the system clock.

RETURN VALUE

If successful, the *pthread_condattr_getclock*( ) function shall return zero and store the value of the clock attribute of *attr* into the object referenced by the *clock_id* argument. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_condattr_setclock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_condattr_setclock*( ) function may fail if:

[EINVAL]	The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_init( ), *pthread_cond_timedwait*( ), *pthread_condattr_destroy*( ), *pthread_condattr_getpshared*( ) (on page 1536),1 *pthread_condattr_init*( ), *pthread_condattr_setpshared*( ) (on page 1540),1 *pthread_create*( ), *pthread_mutex_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

pthread_condattr_getpshared, pthread_condattr_setpshared — get and set the process-shared condition variable attributes

SYNOPSIS

THR TSH

```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
    int *restrict pshared);
int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```

DESCRIPTION

The *pthread_condattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute from the attributes object referenced by *attr*. The *pthread_condattr_setpshared*( ) function shall set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated, even if the condition variable is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the condition variable shall only be operated upon by threads created within the same process as the thread that initialized the condition variable; if threads of differing processes attempt to operate on such a condition variable, the behavior is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

RETURN VALUE

If successful, the *pthread_condattr_setpshared*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_condattr_getpshared*( ) function shall return zero and store the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_condattr_setpshared*( ) function may fail if:

[EINVAL]	The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create( ), *pthread_cond_destroy*( ), *pthread_condattr_destroy*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_condattr_getpshared*( ) and *pthread_condattr_setpshared*( ) functions are marked as part of the Threads and Thread Process-Shared Synchronization options.

The **restrict** keyword is added to the *pthread_condattr_getpshared*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

pthread_condattr_init — initialize condition variable attributes object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
```

DESCRIPTION

Refer to *pthread_condattr_destroy*( ).

NAME

pthread_condattr_setclock — set the clock selection condition variable attribute

SYNOPSIS

THR CS

```
#include <pthread.h>

int pthread_condattr_setclock(pthread_condattr_t *attr,
    clockid_t clock_id);
```

DESCRIPTION

Refer to *pthread_condattr_getclock*( ).

NAME

pthread_condattr_setpshared — set the process-shared condition variable attributes

SYNOPSIS

THR TSH

```
#include <pthread.h>

int pthread_condattr_setpshared(pthread_condattr_t *attr,
    int pshared);
```

DESCRIPTION

Refer to *pthread_condattr_getpshared*( ).

NAME

pthread_create — thread creation

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
```

DESCRIPTION

The *pthread_create*( ) function shall create a new thread, with attributes specified by *attr*, within a process. If *attr* is NULL, the default attributes shall be used. If the attributes specified by *attr* are modified later, the thread's attributes shall not be affected. Upon successful completion, *pthread_create*( ) shall store the ID of the created thread in the location referenced by *thread*.

The thread is created executing *start_routine* with *arg* as its sole argument. If the *start_routine* returns, the effect shall be as if there was an implicit call to *pthread_exit*( ) using the return value of *start_routine* as the exit status. Note that the thread in which *main*( ) was originally invoked differs from this. When it returns from *main*( ), the effect shall be as if there was an implicit call to *exit*( ) using the return value of *main*( ) as the exit status.

The signal state of the new thread shall be initialized as follows:

- The signal mask shall be inherited from the creating thread.
- The set of signals pending for the new thread shall be empty.

The floating-point environment shall be inherited from the creating thread.

If *pthread_create*( ) fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

TCT If _POSIX_THREAD_CPUTIME is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

RETURN VALUE

If successful, the *pthread_create*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_create*( ) function shall fail if:

[EAGAIN]	The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process PTHREAD_THREADS_MAX would be exceeded.
[EINVAL]	The value specified by *attr* is invalid.
[EPERM]	The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

The *pthread_create*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A suggested alternative to *pthread_create*( ) would be to define two separate operations: create and start. Some applications would find such behavior more natural. Ada, in particular, separates the ''creation'' of a task from its ''activation''.

Splitting the operation was rejected by the standard developers for many reasons:

- The number of calls required to start a thread would increase from one to two and thus place an additional burden on applications that do not require the additional synchronization. The second call, however, could be avoided by the additional complication of a start-up state attribute.
- An extra state would be introduced: ''created but not started''. This would require the standard to specify the behavior of the thread operations when the target has not yet started executing.
- For those applications that require such behavior, it is possible to simulate the two separate steps with the facilities that are currently provided. The *start_routine*( ) can synchronize by waiting on a condition variable that is signaled by the start operation.

An Ada implementor can choose to create the thread at either of two points in the Ada program: when the task object is created, or when the task is activated (generally at a ''begin''). If the first approach is adopted, the *start_routine*( ) needs to wait on a condition variable to receive the order to begin ''activation''. The second approach requires no such condition variable or extra synchronization. In either approach, a separate Ada task control block would need to be created when the task object is created to hold rendezvous queues, and so on.

An extension of the preceding model would be to allow the state of the thread to be modified between the create and start. This would allow the thread attributes object to be eliminated. This has been rejected because:

- All state in the thread attributes object has to be able to be set for the thread. This would require the definition of functions to modify thread attributes. There would be no reduction in the number of function calls required to set up the thread. In fact, for an application that creates all threads using identical attributes, the number of function calls required to set up the threads would be dramatically increased. Use of a thread attributes object permits the application to make one set of attribute setting function calls. Otherwise, the set of attribute setting function calls needs to be made for each thread creation.
- Depending on the implementation architecture, functions to set thread state would require kernel calls, or for other implementation reasons would not be able to be implemented as macros, thereby increasing the cost of thread creation.
- The ability for applications to segregate threads by class would be lost.

Another suggested alternative uses a model similar to that for process creation, such as ''thread fork''. The fork semantics would provide more flexibility and the ''create'' function can be implemented simply by doing a thread fork followed immediately by a call to the desired ''start routine'' for the thread. This alternative has these problems:

- For many implementations, the entire stack of the calling thread would need to be duplicated, since in many architectures there is no way to determine the size of the calling frame.

- Efficiency is reduced since at least some part of the stack has to be copied, even though in most cases the thread never needs the copied context, since it merely calls the desired start routine.

FUTURE DIRECTIONS

None.

SEE ALSO

fork( ), *pthread_exit*( ), *pthread_join*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_create*( ) function is marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EPERM] mandatory error condition is added.

The thread CPU-time clock semantics are added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_create*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

The DESCRIPTION is updated to make it explicit that the floating-point environment is inherited from the creating thread.

NAME

pthread_detach — detach a thread

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

DESCRIPTION

The *pthread_detach*( ) function shall indicate to the implementation that storage for the thread *thread* can be reclaimed when that thread terminates. If *thread* has not terminated, *pthread_detach*( ) shall not cause it to terminate. The effect of multiple *pthread_detach*( ) calls on the same target thread is unspecified. |

RETURN VALUE

If the call succeeds, *pthread_detach*( ) shall return 0; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_detach*( ) function shall fail if:

[EINVAL]	The implementation has detected that the value specified by *thread* does not refer to a joinable thread.
[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.

The *pthread_detach*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pthread_join*( ) or *pthread_detach*( ) functions should eventually be called for every thread that is created so that storage associated with the thread may be reclaimed.

It has been suggested that a ‘‘detach’’ function is not necessary; the *detachstate* thread creation attribute is sufficient, since a thread need never be dynamically detached. However, need arises in at least two cases:

1. In a cancelation handler for a *pthread_join*( ) it is nearly essential to have a *pthread_detach*( ) function in order to detach the thread on which *pthread_join*( ) was waiting. Without it, it would be necessary to have the handler do another *pthread_join*( ) to attempt to detach the thread, which would both delay the cancelation processing for an unbounded period and introduce a new call to *pthread_join*( ), which might itself need a cancelation handler. A dynamic detach is nearly essential in this case.
2. In order to detach the ‘‘initial thread’’ (as may be desirable in processes that set up server threads).

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_join( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_detach*( ) function is marked as part of the Threads option.

NAME

pthread_equal — compare thread IDs

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

DESCRIPTION

This function shall compare the thread IDs *t1* and *t2*.

RETURN VALUE

The *pthread_equal*( ) function shall return a non-zero value if *t1* and *t2* are equal; otherwise, zero shall be returned.

If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

ERRORS

No errors are defined.

The *pthread_equal*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Implementations may choose to define a thread ID as a structure. This allows additional flexibility and robustness over using an **int**. For example, a thread ID could include a sequence number that allows detection of ‘‘dangling IDs’’ (copies of a thread ID that has been detached). Since the C language does not support comparison on structure types, the *pthread_equal*( ) function is provided to compare thread IDs.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create( ), *pthread_self*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_equal*( ) function is marked as part of the Threads option.

NAME

pthread_exit — thread termination

SYNOPSIS

THR

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
```

DESCRIPTION

The *pthread_exit*( ) function shall terminate the calling thread and make the value *value_ptr* available to any successful join with the terminating thread. Any cancelation cleanup handlers that have been pushed and not yet popped shall be popped in the reverse order that they were pushed and then executed. After all cancelation cleanup handlers have been executed, if the thread has any thread-specific data, appropriate destructor functions shall be called in an unspecified order. Thread termination does not release any application visible process resources, including, but not limited to, mutexes and file descriptors, nor does it perform any process-level cleanup actions, including, but not limited to, calling any *atexit*( ) routines that may exist.

An implicit call to *pthread_exit*( ) is made when a thread other than the thread in which *main*( ) was first invoked returns from the start routine that was used to create it. The function's return value shall serve as the thread's exit status.

The behavior of *pthread_exit*( ) is undefined if called from a cancelation cleanup handler or destructor function that was invoked as a result of either an implicit or explicit call to *pthread_exit*( ).

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. Thus, references to local variables of the exiting thread should not be used for the *pthread_exit*( ) *value_ptr* parameter value.

The process shall exit with an exit status of 0 after the last thread has been terminated. The behavior shall be as if the implementation called *exit*( ) with a zero argument at thread termination time.

RETURN VALUE

The *pthread_exit*( ) function cannot return to its caller.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The normal mechanism by which a thread terminates is to return from the routine that was specified in the *pthread_create*( ) call that started it. The *pthread_exit*( ) function provides the capability for a thread to terminate without requiring a return from the start routine of that thread, thereby providing a function analogous to *exit*( ).

Regardless of the method of thread termination, any cancelation cleanup handlers that have been pushed and not yet popped are executed, and the destructors for any existing thread-specific data are executed. This volume of IEEE Std 1003.1-200x requires that cancelation cleanup handlers be popped and called in order. After all cancelation cleanup handlers have been executed, thread-specific data destructors are called, in an unspecified order, for each item of thread-specific data that exists in the thread. This ordering is necessary because cancelation cleanup handlers may rely on thread-specific data.

As the meaning of the status is determined by the application (except when the thread has been canceled, in which case it is PTHREAD_CANCELED), the implementation has no idea what an illegal status value is, which is why no address error checking is done.

FUTURE DIRECTIONS

None.

SEE ALSO

exit( ), *pthread_create*( ), *pthread_join*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_exit*( ) function is marked as part of the Threads option.

NAME

pthread_getconcurrency, pthread_setconcurrency — get and set level of concurrency

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_getconcurrency(void);
int pthread_setconcurrency(int new_level);

```

DESCRIPTION

Unbound threads in a process may or may not be required to be simultaneously active. By default, the threads implementation ensures that a sufficient number of threads are active so that the process can continue to make progress. While this conserves system resources, it may not produce the most effective level of concurrency.

The *pthread_setconcurrency*( ) function allows an application to inform the threads implementation of its desired concurrency level, *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If *new_level* is zero, it causes the implementation to maintain the concurrency level at its discretion as if *pthread_setconcurrency*( ) had never been called.

The *pthread_getconcurrency*( ) function shall return the value set by a previous call to the *pthread_setconcurrency*( ) function. If the *pthread_setconcurrency*( ) function was not previously called, this function shall return zero to indicate that the implementation is maintaining the concurrency level.

A call to *pthread_setconcurrency*( ) shall inform the implementation of its desired concurrency level. The implementation shall use this as a hint, not a requirement.

If an implementation does not support multiplexing of user threads on top of several kernel-scheduled entities, the *pthread_setconcurrency*( ) and *pthread_getconcurrency*( ) functions are provided for source code compatibility but they shall have no effect when called. To maintain the function semantics, the *new_level* parameter is saved when *pthread_setconcurrency*( ) is called so that a subsequent call to *pthread_getconcurrency*( ) shall return the same value.

RETURN VALUE

If successful, the *pthread_setconcurrency*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_getconcurrency*( ) function shall always return the concurrency level set by a previous call to *pthread_setconcurrency*( ). If the *pthread_setconcurrency*( ) function has never been called, *pthread_getconcurrency*( ) shall return zero.

ERRORS

The *pthread_setconcurrency*( ) function shall fail if:

[EINVAL]	The value specified by *new_level* is negative.
[EAGAIN]	The value specific by *new_level* would cause a system resource to be exceeded.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Use of these functions changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the *pthread_getconcurrency*( ) and *pthread_setconcurrency*( ) functions since their use may conflict with an applications use of these functions.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

NAME

pthread_getcpuclockid — access a thread CPU-time clock (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR TCT

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

DESCRIPTION

The *pthread_getcpuclockid*( ) function shall return in *clock_id* the clock ID of the CPU-time clock of the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

RETURN VALUE

Upon successful completion, *pthread_getcpuclockid*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_getcpuclockid*( ) function may fail if:

[ESRCH] The value specified by *thread_id* does not refer to an existing thread.

EXAMPLES

None.

APPLICATION USAGE

The *pthread_getcpuclockid*( ) function is part of the Thread CPU-Time Clocks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getcpuclockid( ), *clock_getres*( ), *timer_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_getschedparam, pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

THR TPS

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread, int *restrict policy,
    struct sched_param *restrict param);
int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

DESCRIPTION

The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions shall, respectively, get and set the scheduling policy and parameters of individual threads within a multi-threaded process to be retrieved and set. For SCHED_FIFO and SCHED_RR, the only required member of the **sched_param** structure is the priority *sched_priority*. For SCHED_OTHER, the affected scheduling parameters are implementation-defined.

The *pthread_getschedparam*( ) function shall retrieve the scheduling policy and scheduling parameters for the thread whose thread ID is given by *thread* and shall store those values in *policy* and *param*, respectively. The priority value returned from *pthread_getschedparam*( ) shall be the value specified by the most recent *pthread_setschedparam*( ), *pthread_setschedprio*( ), or *pthread_create*( ) call affecting the target thread. It shall not reflect any temporary adjustments to its priority as a result of any priority inheritance or ceiling functions. The *pthread_setschedparam*( ) function shall set the scheduling policy and associated scheduling parameters for the thread whose thread ID is given by *thread* to the policy and associated parameters provided in *policy* and *param*, respectively.

The *policy* parameter may have the value SCHED_OTHER, SCHED_FIFO, or SCHED_RR. The scheduling parameters for the SCHED_OTHER policy are implementation-defined. The SCHED_FIFO and SCHED_RR policies shall have a single scheduling parameter, *priority*.

TSP If _POSIX_THREAD_SPORADIC_SERVER is defined, then the *policy* argument may have the value SCHED_SPORADIC, with the exception for the *pthread_setschedparam*( ) function that if the scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation-defined whether the function is supported; in other words, the implementation need not allow the application to dynamically change the scheduling policy to SCHED_SPORADIC. The sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the *pthread_setschedparam*( ) function fails, the scheduling parameters shall not be changed for the target thread.

RETURN VALUE

If successful, the *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_getschedparam*( ) function may fail if:

[ESRCH]	The value specified by *thread* does not refer to a existing thread.

The *pthread_setschedparam*( ) function may fail if:

[EINVAL]	The value specified by *policy* or one of the scheduling parameters associated with the scheduling policy *policy* is invalid.
[ENOTSUP]	An attempt was made to set the policy or scheduling parameters to an unsupported value.
TSP [ENOTSUP]	An attempt was made to dynamically change the scheduling policy to SCHED_SPORADIC, and the implementation does not support this change.
[EPERM]	The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.
[EPERM]	The implementation does not allow the application to modify one of the parameters to the value specified.
[ESRCH]	The value specified by *thread* does not refer to a existing thread.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_setschedprio( ), *sched_getparam*( ), *sched_getscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<sched.h>** |

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_getschedparam*( ) and *pthread_setschedparam*( ) functions are marked as part of the Threads and Thread Execution Scheduling options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Execution Scheduling option.

The Open Group Corrigendum U026/2 is applied, correcting the prototype for the *pthread_setschedparam*( ) function so that its second argument is of type **int**.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_getschedparam*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Corrigendum U047/1 is applied. |

IEEE PASC Interpretation 1003.1 #96 is applied, noting that priority values can also be set by a call to the *pthread_setschedprio*( ) function.

NAME

pthread_getspecific, pthread_setspecific — thread-specific data management

SYNOPSIS

THR

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

DESCRIPTION

The *pthread_getspecific*( ) function shall return the value currently bound to the specified *key* on behalf of the calling thread.

The *pthread_setspecific*( ) function shall associate a thread-specific *value* with a *key* obtained via a previous call to *pthread_key_create*( ). Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The effect of calling *pthread_getspecific*( ) or *pthread_setspecific*( ) with a *key* value not obtained from *pthread_key_create*( ) or after *key* has been deleted with *pthread_key_delete*( ) is undefined.

Both *pthread_getspecific*( ) and *pthread_setspecific*( ) may be called from a thread-specific data destructor function. A call to *pthread_getspecific*( ) for the thread-specific data key being destroyed shall return the value NULL, unless the value is changed (after the destructor starts) by a call to *pthread_setspecific*( ). Calling *pthread_setspecific*( ) from a thread-specific data destructor routine may result either in lost storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

Both functions may be implemented as macros.

RETURN VALUE

The *pthread_getspecific*( ) function shall return the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, then the value NULL shall be returned.

If successful, the *pthread_setspecific*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

No errors are returned from *pthread_getspecific*( ).

The *pthread_setspecific*( ) function shall fail if:

[ENOMEM]	Insufficient memory exists to associate the value with the key.

The *pthread_setspecific*( ) function may fail if:

[EINVAL]	The key value is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Performance and ease-of-use of *pthread_getspecific*( ) is critical for functions that rely on maintaining state in thread-specific data. Since no errors are required to be detected by it, and since the only error that could be detected is the use of an invalid key, the function to *pthread_getspecific*( ) has been designed to favor speed and simplicity over error reporting.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_key_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_getspecific*( ) and *pthread_setspecific*( ) functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #3 (Part 6) is applied, updating the DESCRIPTION.

NAME

pthread_join — wait for thread termination

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

DESCRIPTION

The *pthread_join*( ) function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated. On return from a successful *pthread_join*( ) call with a non-NULL *value_ptr* argument, the value passed to *pthread_exit*( ) by the terminating thread shall be made available in the location referenced by *value_ptr*. When a *pthread_join*( ) returns successfully, the target thread has been terminated. The results of multiple simultaneous calls to *pthread_join*( ) specifying the same target thread are undefined. If the thread calling *pthread_join*( ) is canceled, then the target thread shall not be detached.

It is unspecified whether a thread that has exited but remains unjoined counts against _PTHREAD_THREADS_MAX.

RETURN VALUE

If successful, the *pthread_join*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_join*( ) function shall fail if:

[EINVAL]	The implementation has detected that the value specified by *thread* does not refer to a joinable thread.
[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.

The *pthread_join*( ) function may fail if:

[EDEADLK]	A deadlock was detected or the value of *thread* specifies the calling thread.

The *pthread_join*( ) function shall not return an error code of [EINTR].

EXAMPLES

An example of thread creation and deletion follows:

```
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;

    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

main()
{
    int        ar[1000000];
```

```
        pthread_t  th1, th2;
        subarray   sb1, sb2;

        sb1.ar = &ar[0];
        sb1.n  = 500000;
        (void) pthread_create(&th1, NULL, incer, &sb1);

        sb2.ar = &ar[500000];
        sb2.n  = 500000;
        (void) pthread_create(&th2, NULL, incer, &sb2);

        (void) pthread_join(th1, NULL);
        (void) pthread_join(th2, NULL);
    }
```

APPLICATION USAGE

None.

RATIONALE

The *pthread_join*( ) function is a convenience that has proven useful in multi-threaded applications. It is true that a programmer could simulate this function if it were not provided by passing extra state as part of the argument to the *start_routine*( ). The terminating thread would set a flag to indicate termination and broadcast a condition that is part of that state; a joining thread would wait on that condition variable. While such a technique would allow a thread to wait on more complex conditions (for example, waiting for multiple threads to terminate), waiting on individual thread termination is considered widely useful. Also, including the *pthread_join*( ) function in no way precludes a programmer from coding such complex waits. Thus, while not a primitive, including *pthread_join*( ) in this volume of IEEE Std 1003.1-200x was considered valuable.

The *pthread_join*( ) function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up resources that were used by the thread. For instance, after *pthread_join*( ) returns, any application-provided stack storage could be reclaimed.

The *pthread_join*( ) or *pthread_detach*( ) function should eventually be called for every thread that is created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE so that storage associated with the thread may be reclaimed.

The interaction between *pthread_join*( ) and cancelation is well-defined for the following reasons:

- The *pthread_join*( ) function, like all other non-async-cancel-safe functions, can only be called with deferred cancelability type.
- Cancelation cannot occur in the disabled cancelability state.

Thus, only the default cancelability state need be considered. As specified, either the *pthread_join*( ) call is canceled, or it succeeds, but not both. The difference is obvious to the application, since either a cancelation handler is run or *pthread_join*( ) returns. There are no race conditions since *pthread_join*( ) was called in the deferred cancelability state.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_join*( ) function is marked as part of the Threads option.

NAME

pthread_key_create — thread-specific data key creation

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

DESCRIPTION

The *pthread_key_create*( ) function shall create a thread-specific data key visible to all threads in the process. Key values provided by *pthread_key_create*( ) are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by *pthread_setspecific*( ) are maintained on a per-thread basis and persist for the life of the calling thread.

Upon key creation, the value NULL shall be associated with the new key in all active threads. Upon thread creation, the value NULL shall be associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the value of the key is set to NULL, and then the function pointed to is called with the previously associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If, after all the destructors have been called for all non-NULL values with associated destructors, there are still some non-NULL values with associated destructors, then the process is repeated. If, after at least {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls for outstanding non-NULL values, there are still some non-NULL values with associated destructors, implementations may stop calling destructors, or they may continue calling destructors until no non-NULL values with associated destructors exist, even though this might result in an infinite loop.

RETURN VALUE

If successful, the *pthread_key_create*( ) function shall store the newly created key value at **key* and shall return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_key_create*( ) function shall fail if:

[EAGAIN]	The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process PTHREAD_KEYS_MAX has been exceeded.
[ENOMEM]	Insufficient memory exists to create the key.

The *pthread_key_create*( ) function shall not return an error code of [EINTR].

EXAMPLES

The following example demonstrates a function that initializes a thread-specific data key when it is first called, and associates a thread-specific object with each calling thread, initializing this object when necessary.

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}

func()
{
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

Note that the key has to be initialized before *pthread_getspecific*( ) or *pthread_setspecific*( ) can be used. The *pthread_key_create*( ) call could either be explicitly made in a module initialization routine, or it can be done implicitly by the first call to a module as in this example. Any attempt to use the key before it is initialized is a programming error, making the code below incorrect.

```
static pthread_key_t key;

func()
{
    void *ptr;

   /* KEY NOT INITIALIZED!!!  THIS WON'T WORK!!! */
    if ((ptr = pthread_getspecific(key)) == NULL &&
        pthread_setspecific(key, NULL) != 0) {
        pthread_key_create(&key, NULL);
        ...
    }
}
```

APPLICATION USAGE

None.

RATIONALE

Destructor Functions

Normally, the value bound to a key on behalf of a particular thread is a pointer to storage allocated dynamically on behalf of the calling thread. The destructor functions specified with *pthread_key_create*( ) are intended to be used to free this storage when the thread exits. Thread cancelation cleanup handlers cannot be used for this purpose because thread-specific data may persist outside the lexical scope in which the cancelation cleanup handlers operate.

If the value associated with a key needs to be updated during the lifetime of the thread, it may be necessary to release the storage associated with the old value before the new value is bound. Although the *pthread_setspecific*( ) function could do this automatically, this feature is not needed often enough to justify the added complexity. Instead, the programmer is responsible for freeing the stale storage:

```
pthread_getspecific(key, &old);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

Note: The above example could leak storage if run with asynchronous cancelation enabled. No such problems occur in the default cancelation state if no cancelation points occur between the get and set.

There is no notion of a destructor-safe function. If an application does not call *pthread_exit*( ) from a signal handler, or if it blocks any signal whose handler may call *pthread_exit*( ) while calling async-unsafe functions, all functions may be safely called from destructors.

Non-Idempotent Data Key Creation

There were requests to make *pthread_key_create*( ) idempotent with respect to a given *key* address parameter. This would allow applications to call *pthread_key_create*( ) multiple times for a given *key* address and be guaranteed that only one key would be created. Doing so would require the key value to be previously initialized (possibly at compile time) to a known null value and would require that implicit mutual-exclusion be performed based on the address and contents of the *key* parameter in order to guarantee that exactly one key would be created.

Unfortunately, the implicit mutual-exclusion would not be limited to only *pthread_key_create*( ). On many implementations, implicit mutual-exclusion would also have to be performed by *pthread_getspecific*( ) and *pthread_setspecific*( ) in order to guard against using incompletely stored or not-yet-visible key values. This could significantly increase the cost of important operations, particularly *pthread_getspecific*( ).

Thus, this proposal was rejected. The *pthread_key_create*( ) function performs no implicit synchronization. It it the responsibility of the programmer to ensure that it is called exactly once per key before use of the key. Several straightforward mechanisms can already be used to accomplish this, including calling explicit module initialization functions, using mutexes, and using *pthread_once*( ). This places no significant burden on the programmer, introduces no possibly confusing *ad hoc* implicit synchronization mechanism, and potentially allows commonly used thread-specific data operations to be more efficient.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_getspecific( ), *pthread_key_delete*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_key_create*( ) function is marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #8 is applied, updating the DESCRIPTION.

NAME

pthread_key_delete — thread-specific data key deletion

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

DESCRIPTION

The *pthread_key_delete*( ) function shall delete a thread-specific data key previously returned by *pthread_key_create*( ). The thread-specific data values associated with *key* need not be NULL at the time *pthread_key_delete*( ) is called. It is the responsibility of the application to free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads; this cleanup can be done either before or after *pthread_key_delete*( ) is called. Any attempt to use *key* following the call to *pthread_key_delete*( ) results in undefined behavior.

The *pthread_key_delete*( ) function shall be callable from within destructor functions. No destructor functions shall be invoked by *pthread_key_delete*( ). Any destructor function that may have been associated with *key* shall no longer be called upon thread exit.

RETURN VALUE

If successful, the *pthread_key_delete*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_key_delete*( ) function may fail if:

[EINVAL]	The *key* value is invalid.

The *pthread_key_delete*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A thread-specific data key deletion function has been included in order to allow the resources associated with an unused thread-specific data key to be freed. Unused thread-specific data keys can arise, among other scenarios, when a dynamically loaded module that allocated a key is unloaded.

Conforming applications are responsible for performing any cleanup actions needed for data structures associated with the key to be deleted, including data referenced by thread-specific data values. No such cleanup is done by *pthread_key_delete*( ). In particular, destructor functions are not called. There are several reasons for this division of responsibility: |

1. The associated destructor functions used to free thread-specific data at thread exit time are only guaranteed to work correctly when called in the thread that allocated the thread-specific data. (Destructors themselves may utilize thread-specific data.) Thus, they cannot be used to free thread-specific data in other threads at key deletion time. Attempting to have them called by other threads at key deletion time would require other threads to be asynchronously interrupted. But since interrupted threads could be in an arbitrary state, including holding locks necessary for the destructor to run, this approach would fail. In general, there is no safe mechanism whereby an implementation could free thread-specific data at key deletion time.

2. Even if there were a means of safely freeing thread-specific data associated with keys to be deleted, doing so would require that implementations be able to enumerate the threads with non-NULL data and potentially keep them from creating more thread-specific data while the key deletion is occurring. This special case could cause extra synchronization in the normal case, which would otherwise be unnecessary.

For an application to know that it is safe to delete a key, it has to know that all the threads that might potentially ever use the key do not attempt to use it again. For example, it could know this if all the client threads have called a cleanup procedure declaring that they are through with the module that is being shut down, perhaps by zero'ing a reference count.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_key_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_key_delete*( ) function is marked as part of the Threads option.

NAME

pthread_kill — send a signal to a thread

SYNOPSIS

THR

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

DESCRIPTION

The *pthread_kill*( ) function shall request that a signal be delivered to the specified thread. |

As in *kill*( ), if *sig* is zero, error checking shall be performed but no signal shall actually be sent. |

RETURN VALUE

Upon successful completion, the function shall return a value of zero. Otherwise, the function shall return an error number. If the *pthread_kill*( ) function fails, no signal shall be sent.

ERRORS

The *pthread_kill*( ) function shall fail if:

[ESRCH]	No thread could be found corresponding to that specified by the given thread ID.
[EINVAL]	The value of the *sig* argument is an invalid or unsupported signal number.

The *pthread_kill*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_kill*( ) function provides a mechanism for asynchronously directing a signal at a thread in the calling process. This could be used, for example, by one thread to affect broadcast delivery of a signal to a set of threads.

Note that *pthread_kill*( ) only causes the signal to be handled in the context of the given thread; the signal action (termination or stopping) affects the process as a whole.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

kill( ), *pthread_self*( ), *raise*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_kill*( ) function is marked as part of the Threads option.

The APPLICATION USAGE section is added.

NAME

pthread_mutex_destroy, pthread_mutex_init — destroy and initialize a mutex

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

The *pthread_mutex_destroy*( ) function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*( ) to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be reinitialized using *pthread_mutex_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

It shall be safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

The *pthread_mutex_init*( ) function shall initialize the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Only *mutex* itself may be used for performing synchronization. The result of referring to copies of *mutex* in calls to *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), *pthread_mutex_unlock*( ), and *pthread_mutex_destroy*( ) is undefined.

Attempting to initialize an already initialized mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect shall be equivalent to dynamic initialization by a call to *pthread_mutex_init*( ) with parameter *attr* specified as NULL, except that no error checks are performed.

RETURN VALUE

If successful, the *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and shall cause an error return prior to modifying the state of the mutex specified by *mutex*.

ERRORS

The *pthread_mutex_destroy*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a *pthread_cond_timedwait*( ) or *pthread_cond_wait*( )) by another thread.
[EINVAL]	The value specified by *mutex* is invalid.

The *pthread_mutex_init*( ) function shall fail if:

[EAGAIN]	The system lacked the necessary resources (other than memory) to initialize another mutex.

[ENOMEM]	Insufficient memory exists to initialize the mutex.
[EPERM]	The caller does not have the privilege to perform the operation.

The *pthread_mutex_init*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by *mutex*, a previously initialized, but not yet destroyed, mutex.
[EINVAL]	The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Alternate Implementations Possible

This volume of IEEE Std 1003.1-200x supports several alternative implementations of mutexes. An implementation may store the lock directly in the object of type **pthread_mutex_t**. Alternatively, an implementation may store the lock in the heap and merely store a pointer, handle, or unique ID in the mutex object. Either implementation has advantages or may be required on certain hardware configurations. So that portable code can be written that is invariant to this choice, this volume of IEEE Std 1003.1-200x does not define assignment or equality for this type, and it uses the term ‘‘initialize’’ to reinforce the (more restrictive) notion that the lock may actually reside in the mutex object itself.

Note that this precludes an over-specification of the type of the mutex or condition variable and motivates the opacity of the type.

An implementation is permitted, but not required, to have *pthread_mutex_destroy*( ) store an illegal value into the mutex. This may help detect erroneous programs that try to lock (or otherwise reference) a mutex that has already been destroyed.

Tradeoff Between Error Checks and Performance Supported

Many of the error checks were made optional in order to let implementations trade off performance *versus* degree of error checking according to the needs of their specific applications and execution environment. As a general rule, errors or conditions caused by the system (such as insufficient memory) always need to be reported, but errors due to an erroneously coded application (such as failing to provide adequate synchronization to prevent a mutex from being deleted while in use) are made optional.

A wide range of implementations is thus made possible. For example, an implementation intended for application debugging may implement all of the error checks, but an implementation running a single, provably correct application under very tight performance constraints in an embedded computer might implement minimal checks. An implementation might even be provided in two versions, similar to the options that compilers provide: a full-checking, but slower version; and a limited-checking, but faster version. To forbid this optionality would be a disservice to users.

By carefully limiting the use of ‘‘undefined behavior’’ only to things that an erroneous (badly coded) application might do, and by defining that resource-not-available errors are mandatory, this volume of IEEE Std 1003.1-200x ensures that a fully-conforming application is portable across the full range of implementations, while not forcing all implementations to add overhead to check for numerous things that a correct program never does.

Why No Limits Defined

Defining symbols for the maximum number of mutexes and condition variables was considered but rejected because the number of these objects may change dynamically. Furthermore, many implementations place these objects into application memory; thus, there is no explicit maximum.

Static Initializers for Mutexes and Condition Variables

Providing for static initialization of statically allocated synchronization objects allows modules with private static synchronization variables to avoid runtime initialization tests and overhead. Furthermore, it simplifies the coding of self-initializing modules. Such modules are common in C libraries, where for various reasons the design calls for self-initialization instead of requiring an explicit module initialization function to be called. An example use of static initialization follows.

Without static initialization, a self-initializing routine *foo*( ) might look as follows:

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

With static initialization, the same routine could be coded as follows:

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;

void foo()
{
    pthread_mutex_lock(&foo_mutex);
   /* Do work. */
    pthread_mutex_unlock(&foo_mutex);
}
```

Note that the static initialization both eliminates the need for the initialization test inside *pthread_once*( ) and the fetch of &*foo_mutex* to learn the address to be passed to *pthread_mutex_lock*( ) or *pthread_mutex_unlock*( ).

Thus, the C code written to initialize static objects is simpler on all systems and is also faster on a large class of systems; those where the (entire) synchronization object can be stored in application memory.

Yet the locking performance question is likely to be raised for machines that require mutexes to be allocated out of special memory. Such machines actually have to have mutexes and possibly condition variables contain pointers to the actual hardware locks. For static initialization to work on such machines, *pthread_mutex_lock*( ) also has to test whether or not the pointer to the actual lock has been allocated. If it has not, *pthread_mutex_lock*( ) has to initialize it before use. The reservation of such resources can be made when the program is loaded, and hence return codes have not been added to mutex locking and condition variable waiting to indicate failure to complete initialization.

This runtime test in *pthread_mutex_lock*( ) would at first seem to be extra work; an extra test is required to see whether the pointer has been initialized. On most machines this would actually be implemented as a fetch of the pointer, testing the pointer against zero, and then using the pointer if it has already been initialized. While the test might seem to add extra work, the extra effort of testing a register is usually negligible since no extra memory references are actually done. As more and more machines provide caches, the real expenses are memory references, not instructions executed.

Alternatively, depending on the machine architecture, there are often ways to eliminate *all* overhead in the most important case: on the lock operations that occur *after* the lock has been initialized. This can be done by shifting more overhead to the less frequent operation: initialization. Since out-of-line mutex allocation also means that an address has to be dereferenced to find the actual lock, one technique that is widely applicable is to have static initialization store a bogus value for that address; in particular, an address that causes a machine fault to occur. When such a fault occurs upon the first attempt to lock such a mutex, validity checks can be done, and then the correct address for the actual lock can be filled in. Subsequent lock operations incur no extra overhead since they do not ‘‘fault’’. This is merely one technique that can be used to support static initialization, while not adversely affecting the performance of lock acquisition. No doubt there are other techniques that are highly machine-dependent.

The locking overhead for machines doing out-of-line mutex allocation is thus similar for modules being implicitly initialized, where it is improved for those doing mutex allocation entirely inline. The inline case is thus made much faster, and the out-of-line case is not significantly worse.

Besides the issue of locking performance for such machines, a concern is raised that it is possible that threads would serialize contending for initialization locks when attempting to finish initializing statically allocated mutexes. (Such finishing would typically involve taking an internal lock, allocating a structure, storing a pointer to the structure in the mutex, and releasing the internal lock.) First, many implementations would reduce such serialization by hashing on the mutex address. Second, such serialization can only occur a bounded number of times. In particular, it can happen at most as many times as there are statically allocated synchronization objects. Dynamically allocated objects would still be initialized via *pthread_mutex_init*( ) or *pthread_cond_init*( ).

Finally, if none of the above optimization techniques for out-of-line allocation yields sufficient performance for an application on some implementation, the application can avoid static initialization altogether by explicitly initializing all synchronization objects with the corresponding *pthread_*_init*( ) functions, which are supported by all implementations. An implementation can also document the tradeoffs and advise which initialization technique is more efficient for that particular implementation.

Destroying Mutexes

A mutex can be destroyed immediately after it is unlocked. For example, consider the following code:

```
struct obj {
pthread_mutex_t om;
    int refcnt;
    ...
};

obj_done(struct obj *op)
{
    pthread_mutex_lock(&op->om);
    if (--op->refcnt == 0) {
        pthread_mutex_unlock(&op->om);
(A)     pthread_mutex_destroy(&op->om);
(B)     free(op);
    } else
(C)     pthread_mutex_unlock(&op->om);
}
```

In this case *obj* is reference counted and *obj_done*( ) is called whenever a reference to the object is dropped. Implementations are required to allow an object to be destroyed and freed and potentially unmapped (for example, lines A and B) immediately after the object is unlocked (line C).

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_getprioceiling( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), *pthread_mutexattr_getpshared*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutex_destroy*( ) and *pthread_mutex_init*( ) functions are marked as part of the Threads option.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1c #34 is applied, updating the DESCRIPTION.

The **restrict** keyword is added to the *pthread_mutex_init*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

NAME

pthread_mutex_getprioceiling, pthread_mutex_setprioceiling — get and set the priority ceiling of a mutex (**REALTIME THREADS**)

SYNOPSIS

THR TPP

```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
    int *restrict prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

DESCRIPTION

The *pthread_mutex_getprioceiling*( ) function shall return the current priority ceiling of the mutex.

The *pthread_mutex_setprioceiling*( ) function shall either lock the mutex if it is unlocked, or block until it can successfully lock the mutex, then it shall change the mutex's priority ceiling and release the mutex. When the change is successful, the previous value of the priority ceiling shall be returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.

If the *pthread_mutex_setprioceiling*( ) function fails, the mutex priority ceiling shall not be changed.

RETURN VALUE

If successful, the *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions may fail if:

[EINVAL]	The priority requested by *prioceiling* is out of range.
[EINVAL]	The value specified by *mutex* does not refer to a currently existing mutex.
[EPERM]	The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *pthread_mutex_lock*( ), *pthread_mutex_timedlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOSYS] error denoting non-support of the priority ceiling protocol for mutexes has been removed. This is since if the implementation provides the functions (regardless of whether _POSIX_PTHREAD_PRIO_PROTECT is defined), they must function as in the DESCRIPTION and therefore the priority ceiling protocol for mutexes is supported.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *pthread_mutex_getprioceiling*( ) and *pthread_mutex_setprioceiling*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

NAME

pthread_mutex_init — initialize a mutex

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
    const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

Refer to *pthread_mutex_destroy*( ).

NAME

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*( ). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

XSI If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided. If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The *pthread_mutex_trylock*( ) function shall be equivalent to *pthread_mutex_lock*( ), except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the *pthread_mutex_trylock*( ) function shall immediately return success.

XSI The *pthread_mutex_unlock*( ) function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*( ) is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

XSI (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex shall become available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread shall resume waiting for the mutex as if it was not interrupted.

RETURN VALUE

If successful, the *pthread_mutex_lock*( ) and *pthread_mutex_unlock*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_mutex_trylock*( ) function shall return zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

ERRORS

The *pthread_mutex_lock*( ) and *pthread_mutex_trylock*( ) functions shall fail if:

[EINVAL]	The *mutex* was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex's current priority ceiling.

The *pthread_mutex_trylock*( ) function shall fail if:

[EBUSY]	The *mutex* could not be acquired because it was already locked.

The *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), and *pthread_mutex_unlock*( ) functions may fail if:

[EINVAL]	The value specified by *mutex* does not refer to an initialized mutex object.
XSI [EAGAIN]	The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded.

The *pthread_mutex_lock*( ) function may fail if:

[EDEADLK]	The current thread already owns the mutex.

The *pthread_mutex_unlock*( ) function may fail if:

[EPERM]	The current thread does not own the mutex.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built. As such, the implementation of mutexes should be as efficient as possible, and this has ramifications on the features available at the interface.

The mutex functions and the particular default settings of the mutex attributes have been motivated by the desire to not preclude fast, inlined implementations of mutex locking and unlocking.

For example, deadlocking on a double-lock is explicitly allowed behavior in order to avoid requiring more overhead in the basic mechanism than is absolutely necessary. (More ‘‘friendly’’ mutexes that detect deadlock or that allow multiple locking by the same thread are easily constructed by the user via the other mechanisms provided. For example, *pthread_self*( ) can be used to record mutex ownership.) Implementations might also choose to provide such extended features as options via special mutex attributes.

Since most attributes only need to be checked when a thread is going to be blocked, the use of attributes does not slow the (common) mutex-locking case.

Likewise, while being able to extract the thread ID of the owner of a mutex might be desirable, it would require storing the current thread ID when each mutex is locked, and this could incur unacceptable levels of overhead. Similar arguments apply to a *mutex_tryunlock* operation.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *pthread_mutex_timedlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), and *pthread_mutex_unlock*( ) functions are marked as part of the Threads option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The behavior when attempting to relock a mutex is defined.

The *pthread_mutex_timedlock*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

pthread_mutex_setprioceiling — change the priority ceiling of a mutex (**REALTIME THREADS**)

SYNOPSIS

THR TPP

```
#include <pthread.h>

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
    int prioceiling, int *restrict old_ceiling);
```

DESCRIPTION

Refer to *pthread_mutex_getprioceiling*( ).

NAME

pthread_mutex_timedlock — lock a mutex (**ADVANCED REALTIME**)

SYNOPSIS

THR TMO

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *pthread_mutex_timedlock*( ) function shall lock the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread shall block until the mutex becomes available as in the *pthread_mutex_lock*( ) function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function.

The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

Under no circumstance shall the function fail with a timeout if the mutex can be locked immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex shall be adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

RETURN VALUE

If successful, the *pthread_mutex_timedlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutex_timedlock*( ) function shall fail if:

[EINVAL]	The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex' current priority ceiling.
[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.
[ETIMEDOUT]	The mutex could not be locked before the specified timeout expired.

The *pthread_mutex_timedlock*( ) function may fail if:

[EINVAL]	The value specified by mutex does not refer to an initialized mutex object.

XSI [EAGAIN]	The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded.
[EDEADLK]	The current thread already owns the mutex.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_mutex_timedlock*( ) function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_mutex_destroy( ), *pthread_mutex_lock*( ), *pthread_mutex_trylock*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

pthread_mutex_trylock, pthread_mutex_unlock — lock and unlock a mutex

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

Refer to *pthread_mutex_lock*( ).

NAME

pthread_mutexattr_destroy, pthread_mutexattr_init — destroy and initialize mutex attributes object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

DESCRIPTION

The *pthread_mutexattr_destroy*( ) function shall destroy a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause *pthread_mutexattr_destroy*( ) to set the object referenced by *attr* to an invalid value. A destroyed *attr* attributes object can be reinitialized using *pthread_mutexattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined.

The *pthread_mutexattr_init*( ) function shall initialize a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if *pthread_mutexattr_init*( ) is called specifying an already initialized *attr* attributes object.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) shall not affect any previously initialized mutexes.

RETURN VALUE

Upon successful completion, *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_destroy*( ) function may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_mutexattr_init*( ) function shall fail if:

[ENOMEM]	Insufficient memory exists to initialize the mutex attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

See *pthread_attr_init*( ) for a general explanation of attributes. Attributes objects allow implementations to experiment with useful extensions and permit extension of this volume of IEEE Std 1003.1-200x without changing the existing functions. Thus, they provide for future extensibility of this volume of IEEE Std 1003.1-200x and reduce the temptation to standardize prematurely on semantics that are not yet widely implemented or understood.

Examples of possible additional mutex attributes that have been discussed are *spin_only*, *limited_spin*, *no_spin*, *recursive*, and *metered*. (To explain what the latter attributes might mean: recursive mutexes would allow for multiple re-locking by the current owner; metered mutexes would transparently keep records of queue length, wait time, and so on.) Since there is not yet wide agreement on the usefulness of these resulting from shared implementation and usage experience, they are not yet specified in this volume of IEEE Std 1003.1-200x. Mutex attributes objects, however, make it possible to test out these concepts for possible standardization at a later time.

Mutex Attributes and Performance

Care has been taken to ensure that the default values of the mutex attributes have been defined such that mutexes initialized with the defaults have simple enough semantics so that the locking and unlocking can be done with the equivalent of a test-and-set instruction (plus possibly a few other basic instructions).

There is at least one implementation method that can be used to reduce the cost of testing at lock-time if a mutex has non-default attributes. One such method that an implementation can employ (and this can be made fully transparent to fully conforming POSIX applications) is to secretly pre-lock any mutexes that are initialized to non-default attributes. Any later attempt to lock such a mutex causes the implementation to branch to the ‘‘slow path’’ as if the mutex were unavailable; then, on the slow path, the implementation can do the ‘‘real work’’ to lock a non-default mutex. The underlying unlock operation is more complicated since the implementation never really wants to release the pre-lock on this kind of mutex. This illustrates that, depending on the hardware, there may be certain optimizations that can be used so that whatever mutex attributes are considered ‘‘most frequently used’’ can be processed most efficiently.

Process Shared Memory and Synchronization

The existence of memory mapping functions in this volume of IEEE Std 1003.1-200x leads to the possibility that an application may allocate the synchronization objects from this section in memory that is accessed by multiple processes (and therefore, by threads of multiple processes).

In order to permit such usage, while at the same time keeping the usual case (that is, usage within a single process) efficient, a process-shared option has been defined.

If an implementation supports the _POSIX_THREAD_PROCESS_SHARED option, then the *process-shared* attribute can be used to indicate that mutexes or condition variables may be accessed by threads of multiple processes.

The default setting of PTHREAD_PROCESS_PRIVATE has been chosen for the *process-shared* attribute so that the most efficient forms of these synchronization objects are created by default.

Synchronization variables that are initialized with the PTHREAD_PROCESS_PRIVATE *process-shared* attribute may only be operated on by threads in the process that initialized them. Synchronization variables that are initialized with the PTHREAD_PROCESS_SHARED *process-shared* attribute may be operated on by any thread in any process that has access to it. In particular, these processes may exist beyond the lifetime of the initializing process. For example, the following code implements a simple counting semaphore in a mapped file that may be used by many processes.

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};
typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
```

```
void semaphore_post(semaphore_t *semap);
void semaphore_wait(semaphore_t *semap);
void semaphore_close(semaphore_t *semap);

/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
{
int fd;
    semaphore_t *semap;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;

    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semap->lock, &psharedm);
    (void) pthread_cond_init(&semap->nonzero, &psharedc);
    semap->count = 0;
    return (semap);
}

semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;

    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
            PROT_READ | PROT_WRITE, MAP_SHARED,
            fd, 0);
    close (fd);
    return (semap);
}
```

```
void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
        pthread_cond_signal(&semapx->nonzero);
    semap->count++;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}
```

The following code is for three separate processes that create, post, and wait on a semaphore in the file **/tmp/semaphore**. Once the file is created, the post and wait programs increment and decrement the counting semaphore (waiting and waking as required) even though they did not initialize the semaphore.

```
/* create.c */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;

    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}

/* post */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;
```

```
    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}

/* wait */
#include "pthread.h"
#include "sem.h"

int
main()
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), *pthread_mutexattr_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutexattr_destroy*( ) and *pthread_mutexattr_init*( ) functions are marked as part of the Threads option.

IEEE PASC Interpretation 1003.1c #27 is applied, updating the ERRORS section.

NAME

pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling — get and set prioceiling attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

THR TPP

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(
    const pthread_mutexattr_t *restrict attr,
    int *restrict prioceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```

DESCRIPTION

The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions, respectively, shall get and set the priority ceiling attribute of a mutex attributes object pointed to by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of *prioceiling* are within the maximum range of priorities defined by SCHED_FIFO.

The *prioceiling* attribute defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex shall be set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of *prioceiling* are within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions may fail if:

[EINVAL]	The value specified by *attr* or *prioceiling* is invalid.
[EPERM]	The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutexattr_getprioceiling*( ) and *pthread_mutexattr_setprioceiling*( ) functions are marked as part of the Threads and Thread Priority Protection options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection option.

The [ENOTSUP] error condition has been removed since these functions do not have a *protocol* argument.

The **restrict** keyword is added to the *pthread_mutexattr_getprioceiling*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_mutexattr_getprotocol, pthread_mutexattr_setprotocol — get and set protocol attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

THR
```
#include <pthread.h>
```

TPP|TPI
```
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *restrict attr,
    int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

DESCRIPTION

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions, respectively, shall get and set the protocol attribute of a mutex attributes object pointed to by *attr* which was previously created by the function *pthread_mutexattr_init*( ).

The *protocol* attribute defines the protocol to be followed in utilizing mutexes. The value of *protocol* may be one of:

PTHREAD_PRIO_NONE
TPI PTHREAD_PRIO_INHERIT
TPP PTHREAD_PRIO_PROTECT

which are defined in the **<pthread.h>** header.

When a thread owns a mutex with the PTHREAD_PRIO_NONE *protocol* attribute, its priority and scheduling shall not be affected by its mutex ownership.

TPI When a thread is blocking higher priority threads because of owning one or more mutexes with the PTHREAD_PRIO_INHERIT protocol attribute, it shall execute at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

TPP When a thread owns one or more mutexes initialized with the PTHREAD_PRIO_PROTECT protocol, it shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes or not.

While a thread is holding a mutex which has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to *sched_setparam*( ). Likewise, when a thread unlocks a mutex that has been initialized with the PTHREAD_PRIO_INHERIT or PTHREAD_PRIO_PROTECT protocol attributes, it shall not be subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it shall execute at the highest of the priorities that it would have obtained by each of these protocols.

TPI When a thread makes a call to *pthread_mutex_lock*( ), the mutex was initialized with the protocol attribute having the value PTHREAD_PRIO_INHERIT, when the calling thread is blocked because the mutex is owned by another thread, that owner thread shall inherit the priority level of the calling thread as long as it continues to own the mutex. The implementation shall update its execution priority to the maximum of its assigned priority and all its inherited priorities.

Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority inheritance effect shall be propagated to this other owner thread, in a recursive manner.

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_setprotocol*( ) function shall fail if:

[ENOTSUP]	The value specified by *protocol* is an unsupported value.

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions may fail if:

[EINVAL]	The value specified by *attr* or *protocol* is invalid.
[EPERM]	The caller does not have the privilege to perform the operation.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Marked as part of the Realtime Threads Feature Group.

Issue 6

The *pthread_mutexattr_getprotocol*( ) and *pthread_mutexattr_setprotocol*( ) functions are marked as part of the Threads option and either the Thread Priority Protection or Thread Priority Inheritance options.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Thread Priority Protection or Thread Priority Inheritance options.

The **restrict** keyword is added to the *pthread_mutexattr_getprotocol*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_mutexattr_getpshared, pthread_mutexattr_setpshared — get and set process-shared attribute

SYNOPSIS

THR TSH
```
#include <pthread.h>

int pthread_mutexattr_getpshared(
    const pthread_mutexattr_t *restrict attr,
    int *restrict pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```

DESCRIPTION

The *pthread_mutexattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute from the attributes object referenced by *attr*. The *pthread_mutexattr_setpshared*( ) function shall set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the mutex shall only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute shall be PTHREAD_PROCESS_PRIVATE.

RETURN VALUE

Upon successful completion, *pthread_mutexattr_setpshared*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

Upon successful completion, *pthread_mutexattr_getpshared*( ) shall return zero and stores the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_mutexattr_setpshared*( ) function may fail if:

[EINVAL]	The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_destroy( ), *pthread_create*( ), *pthread_mutex_destroy*( ), *pthread_mutexattr_destroy*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_mutexattr_getpshared*( ) and *pthread_mutexattr_setpshared*( ) functions are marked as part of the Threads and Thread Process-Shared Synchronization options.

The **restrict** keyword is added to the *pthread_mutexattr_getpshared*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_mutexattr_gettype, pthread_mutexattr_settype — get and set a mutex type attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

DESCRIPTION

The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions, respectively, shall get and set the mutex *type* attribute. This attribute is set in the *type* parameter to these functions. The default value of the *type* attribute is PTHREAD_MUTEX_DEFAULT.

The type of mutex is contained in the *type* attribute of the mutex attributes. Valid mutex types include:

PTHREAD_MUTEX_NORMAL

This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it shall deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

PTHREAD_MUTEX_ERRORCHECK

This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it shall return with an error. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

PTHREAD_MUTEX_RECURSIVE

A thread attempting to relock this mutex without first unlocking it shall succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex shall require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked shall return with an error. A thread attempting to unlock an unlocked mutex shall return with an error.

PTHREAD_MUTEX_DEFAULT

Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type which is not locked results in undefined behavior. An implementation may map this mutex to one of the other mutex types.

RETURN VALUE

Upon successful completion, the *pthread_mutexattr_gettype*( ) function shall return zero and store the value of the *type* attribute of *attr* into the object referenced by the *type* parameter. Otherwise, an error shall be returned to indicate the error.

If successful, the *pthread_mutexattr_settype*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_mutexattr_settype*( ) function shall fail if:

[EINVAL] The value *type* is invalid.

The *pthread_mutexattr_gettype*( ) and *pthread_mutexattr_settype*( ) functions may fail if:

[EINVAL] The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a *pthread_cond_timedwait*( ) or *pthread_cond_wait*( ) may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cond_timedwait( ), *pthread_cond_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The Open Group Corrigendum U033/3 is applied. The SYNOPSIS for *pthread_mutexattr_gettype*( ) is updated so that the first argument is of type **const pthread_mutexattr_t ***.

The **restrict** keyword is added to the *pthread_mutexattr_gettype*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

NAME

pthread_mutexattr_init — initialize mutex attributes object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

DESCRIPTION

Refer to *pthread_mutexattr_destroy*( ).

NAME

pthread_mutexattr_setprioceiling — set prioceiling attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

THR TPP

```
#include <pthread.h>

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
    int prioceiling);
```

DESCRIPTION

Refer to *pthread_mutexattr_getprioceiling*( ).

NAME

pthread_mutexattr_setprotocol — set protocol attribute of mutex attributes object (**REALTIME THREADS**)

SYNOPSIS

THR

```
#include <pthread.h>
```

TPP|TPI

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
    int protocol);
```

DESCRIPTION

Refer to *pthread_mutexattr_setprotocol*( ).

NAME

pthread_mutexattr_setpshared — set process-shared attribute

SYNOPSIS

THR TSH

```
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
    int pshared);
```

DESCRIPTION

Refer to *pthread_mutexattr_getpshared*( ).

NAME

pthread_mutexattr_settype — set a mutex type attribute

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

DESCRIPTION

Refer to *pthread_mutexattr_gettype*( ).

NAME

pthread_once — dynamic package initialization

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
    void (*init_routine)(void));
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

DESCRIPTION

The first call to *pthread_once*( ) by any thread in a process, with a given *once_control*, shall call the *init_routine* with no arguments. Subsequent calls of *pthread_once*( ) with the same *once_control* shall not call the *init_routine*. On return from *pthread_once*( ), *init_routine* shall have completed. | The *once_control* parameter shall determine whether the associated initialization routine has | been called.

The *pthread_once*( ) function is not a cancelation point. However, if *init_routine* is a cancelation point and is canceled, the effect on *once_control* shall be as if *pthread_once*( ) was never called.

The constant PTHREAD_ONCE_INIT is defined in the **<pthread.h>** header. |

The behavior of *pthread_once*( ) is undefined if *once_control* has automatic storage duration or is not initialized by PTHREAD_ONCE_INIT.

RETURN VALUE

Upon successful completion, *pthread_once*( ) shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_once*( ) function may fail if:

[EINVAL] If either *once_control* or *init_routine* is invalid.

The *pthread_once*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Some C libraries are designed for dynamic initialization. That is, the global initialization for the library is performed when the first procedure in the library is called. In a single-threaded program, this is normally implemented using a static variable whose value is checked on entry to a routine, as follows:

```
static int random_is_initialized = 0;
extern int initialize_random();

int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ... /* Operations performed after initialization. */
}
```

To keep the same structure in a multi-threaded program, a new primitive is needed. Otherwise, library initialization has to be accomplished by an explicit call to a library-exported initialization function prior to any use of the library.

For dynamic library initialization in a multi-threaded process, a simple initialization flag is not sufficient; the flag needs to be protected against modification by multiple threads simultaneously calling into the library. Protecting the flag requires the use of a mutex; however, mutexes have to be initialized before they are used. Ensuring that the mutex is only initialized once requires a recursive solution to this problem.

The use of *pthread_once*( ) not only supplies an implementation-guaranteed means of dynamic initialization, it provides an aid to the reliable construction of multi-threaded and realtime systems. The preceding example then becomes:

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT;
extern int initialize_random();

int random_function()
{
    (void) pthread_once(&random_is_initialized, initialize_random);
    ... /* Operations performed after initialization. */
}
```

Note that a **pthread_once_t** cannot be an array because some compilers do not accept the construct **&<array_name>**.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_once*( ) function is marked as part of the Threads option.

The [EINVAL] error is added as a may fail case for if either argument is invalid.

NAME

pthread_rwlock_destroy, pthread_rwlock_init — destroy and initialize a read-write lock object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```

DESCRIPTION

The *pthread_rwlock_destroy*( ) function shall destroy the read-write lock object referenced by *rwlock* and release any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to *pthread_rwlock_init*( ). An implementation may cause *pthread_rwlock_destroy*( ) to set the object referenced by *rwlock* to an invalid value. Results are undefined if *pthread_rwlock_destroy*( ) is called when any thread holds *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior.

The *pthread_rwlock_init*( ) function shall allocate any resources required to use the read-write lock referenced by *rwlock* and initializes the lock to an unlocked state with attributes referenced by *attr*. If *attr* is NULL, the default read-write lock attributes shall be used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being reinitialized. Results are undefined if *pthread_rwlock_init*( ) is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

If the *pthread_rwlock_init*( ) function fails, *rwlock* shall not be initialized and the contents of *rwlock* are undefined.

Only the object referenced by *rwlock* may be used for performing synchronization. The result of referring to copies of that object in calls to *pthread_rwlock_destroy*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), or *pthread_rwlock_wrlock*( ) is undefined.

RETURN VALUE

If successful, the *pthread_rwlock_destroy*( ) and *pthread_rwlock_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The [EBUSY] and [EINVAL] error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the read-write lock specified by *rwlock*.

ERRORS

The *pthread_rwlock_destroy*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to destroy the object referenced by *rwlock* while it is locked.
[EINVAL]	The value specified by *rwlock* is invalid.

The *pthread_rwlock_init*( ) function shall fail if:

[EAGAIN]	The system lacked the necessary resources (other than memory) to initialize another read-write lock.
[ENOMEM]	Insufficient memory exists to initialize the read-write lock.
[EPERM]	The caller does not have the privilege to perform the operation.

The *pthread_rwlock_init*( ) function may fail if:

[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by *rwlock*, a previously initialized but not yet destroyed read-write lock.
[EINVAL]	The value specified by *attr* is invalid.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_rdlock( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension). The initializer macro is also deleted from the SYNOPSIS.
- The DESCRIPTION is updated as follows:
  - — It explicitly notes allocation of resources upon initialization of a read-write lock object.
  - — A paragraph is added specifying that copies of read-write lock objects may not be used.
- An [EINVAL] error is added to the ERRORS section for *pthread_rwlock_init*( ), indicating that the *rwlock* value is invalid.
- The SEE ALSO section is updated.

The **restrict** keyword is added to the *pthread_rwlock_init*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_rwlock_init — initialize a read-write lock object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
    const pthread_rwlockattr_t *restrict attr);
```

DESCRIPTION

Refer to *pthread_rwlock_destroy*( ).

NAME

pthread_rwlock_rdlock, pthread_rwlock_tryrdlock — lock a read-write lock object for reading

SYNOPSIS

THR
```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION

The *pthread_rwlock_rdlock*( ) function shall apply a read lock to the read-write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock.

TPS If the Thread Execution Scheduling option is supported, and the threads involved in the lock are executing with the scheduling policies SCHED_FIFO or SCHED_RR, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock.

TPS TSP If the Threads Execution Scheduling option is supported, and the threads involved in the lock are executing with the SCHED_SPORADIC scheduling policy, the calling thread shall not acquire the lock if a writer holds the lock or if writers of higher or equal priority are blocked on the lock; otherwise, the calling thread shall acquire the lock.

If the Thread Execution Scheduling option is not supported, it is implementation-defined whether the calling thread acquires the lock when a writer does not hold the lock and there are writers blocked on the lock. If a writer holds the lock, the calling thread shall not acquire the read lock. If the read lock is not acquired, the calling thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds a write lock.

A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the *pthread_rwlock_rdlock*( ) function *n* times). If so, the application shall ensure that the thread performs matching unlocks (that is, it calls the *pthread_rwlock_unlock*( ) function *n* times).

The maximum number of simultaneous read locks that an implementation guarantees can be applied to a read-write lock shall be implementation-defined. The *pthread_rwlock_rdlock*( ) function may fail if this maximum would be exceeded.

The *pthread_rwlock_tryrdlock*( ) function shall apply a read lock as in the *pthread_rwlock_rdlock*( ) function, with the exception that the function shall fail if the equivalent *pthread_rwlock_rdlock*( ) call would have blocked the calling thread. In no case shall the *pthread_rwlock_tryrdlock*( ) function ever block; it always either acquires the lock or fails and returns immediately.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

RETURN VALUE

If successful, the *pthread_rwlock_rdlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_rwlock_tryrdlock*( ) function shall return zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_tryrdlock*( ) function shall fail if:

[EBUSY] The read-write lock could not be acquired for reading because a writer holds the lock or a writer with the appropriate priority was blocked on it.

The *pthread_rwlock_rdlock*( ) and *pthread_rwlock_tryrdlock*( ) functions may fail if:

[EINVAL] The value specified by *rwlock* does not refer to an initialized read-write lock object.

[EAGAIN] The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

The *pthread_rwlock_rdlock*( ) function may fail if:

[EDEADLK] The current thread already owns the read-write lock for writing.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The DESCRIPTION is updated as follows:
  - Conditions under which writers have precedence over readers are specified.
  - Failure of *pthread_rwlock_tryrdlock*( ) is clarified.
  - A paragraph on the maximum number of read locks is added.
- In the ERRORS sections, [EBUSY] is modified to take into account write priority, and [EDEADLK] is deleted as a *pthread_rwlock_tryrdlock*( ) error.
- The SEE ALSO section is updated.

NAME

pthread_rwlock_timedrdlock — lock a read-write lock for reading

SYNOPSIS

THR TMO

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *pthread_rwlock_timedrdlock*( ) function shall apply a read lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_rdlock*( ) function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call. |

TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header. Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock can be immediately acquired. |

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to *pthread_rwlock_timedrdlock*( ), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds a write lock on *rwlock*. The results are undefined if this function is called with an uninitialized read-write lock.

RETURN VALUE

The *pthread_rwlock_timedrdlock*( ) function shall return zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_timedrdlock*( ) function shall fail if:

[ETIMEDOUT]	The lock could not be acquired before the specified timeout expired.

The *pthread_rwlock_timedrdlock*( ) function may fail if:

[EAGAIN]	The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.
[EDEADLK]	The calling thread already holds a write lock on *rwlock*.
[EINVAL]	The value specified by *rwlock* does not refer to an initialized read-write lock object, or the *abs_timeout* nanosecond value is less than zero or greater than or equal to 1 000 million.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *pthread_rwlock_timedrdlock*( ) function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

pthread_rwlock_timedwrlock — lock a read-write lock for writing

SYNOPSIS

THR TMO

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
    const struct timespec *restrict abs_timeout);
```

DESCRIPTION

The *pthread_rwlock_timedwrlock*( ) function shall apply a write lock to the read-write lock referenced by *rwlock* as in the *pthread_rwlock_wrlock*( ) function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait shall be terminated when the specified timeout expires. The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header. Under no circumstances shall the function fail with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock via a call to *pthread_rwlock_timedwrlock*( ), upon return from the signal handler the thread shall resume waiting for the lock as if it was not interrupted.

The calling thread may deadlock if at the time the call is made it holds the read-write lock. The results are undefined if this function is called with an uninitialized read-write lock.

RETURN VALUE

The *pthread_rwlock_timedwrlock*( ) function shall return zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_timedwrlock*( ) function shall fail if:

[ETIMEDOUT]	The lock could not be acquired before the specified timeout expired.

The *pthread_rwlock_timedwrlock*( ) function may fail if:

[EDEADLK]	The calling thread already holds the *rwlock*.
[EINVAL]	The value specified by rwlock does not refer to an initialized read-write lock object, or the *abs_timeout* nanosecond value is less than zero or greater than or equal to 1 000 million.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *pthread_rwlock_timedwrlock*( ) function is part of the Threads and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_unlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

NAME

pthread_rwlock_tryrdlock — lock a read-write lock object for reading

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION

Refer to *pthread_rwlock_rdlock*( ).

NAME

pthread_rwlock_trywrlock, pthread_rwlock_wrlock — lock a read-write lock object for writing

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

```

DESCRIPTION

The *pthread_rwlock_trywrlock*( ) function shall apply a write lock like the *pthread_rwlock_wrlock*( ) function, with the exception that the function shall fail if any thread currently holds *rwlock* (for reading or writing).

The *pthread_rwlock_wrlock*( ) function shall apply a write lock to the read-write lock referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread shall block until it can acquire the lock. The calling thread may deadlock if at the time the call is made it holds the read-write lock (whether a read or write lock).

Implementations may favor writers over readers to avoid writer starvation.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

RETURN VALUE

The *pthread_rwlock_trywrlock*( ) function shall return zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_rwlock_wrlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_trywrlock*( ) function shall fail if:

[EBUSY]	The read-write lock could not be acquired for writing because it was already locked for reading or writing.

The *pthread_rwlock_trywrlock*( ) and *pthread_rwlock_wrlock*( ) functions may fail if:

[EINVAL]	The value specified by *rwlock* does not refer to an initialized read-write lock object.

The *pthread_rwlock_wrlock*( ) function may fail if:

[EDEADLK]	The current thread already owns the read-write lock for writing or reading.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The [EDEADLK] error is deleted as a *pthread_rwlock_trywrlock*( ) error.
- The SEE ALSO section is updated.

NAME

pthread_rwlock_unlock — unlock a read-write lock object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION

The *pthread_rwlock_unlock*( ) function shall release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If this function is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this function releases the last read lock for this read-write lock object, the read-write lock object shall be put in the unlocked state with no owners.

If this function is called to release a write lock for this read-write lock object, the read-write lock object shall be put in the unlocked state.

If there are threads blocked on the lock when it becomes available, the scheduling policy shall determine which thread(s) shall acquire the lock. TPS If the Thread Execution Scheduling option is supported, when threads executing with the scheduling policies SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on the lock, they shall acquire the lock in priority order when the lock becomes available. For equal priority threads, write locks shall take precedence over read locks. If the Thread Execution Scheduling option is not supported, it is implementation-defined whether write locks take precedence over read locks.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

RETURN VALUE

If successful, the *pthread_rwlock_unlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlock_unlock*( ) function may fail if:

[EINVAL]	The value specified by *rwlock* does not refer to an initialized read-write lock object.
[EPERM]	The current thread does not hold a lock on the read-write lock.

The *pthread_rwlock_unlock*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_destroy( ), *pthread_rwlock_init*( ), *pthread_rwlock_rdlock*( ), *pthread_rwlock_timedrdlock*( ), *pthread_rwlock_timedwrlock*( ), *pthread_rwlock_tryrdlock*( ), *pthread_rwlock_trywrlock*( ), *pthread_rwlock_wrlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The DESCRIPTION is updated as follows:
  - The conditions under which writers have precedence over readers are specified.
  - The concept of read-write lock owner is deleted.
- The SEE ALSO section is updated.

NAME

pthread_rwlock_wrlock — lock a read-write lock object for writing

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

DESCRIPTION

Refer to *pthread_rwlock_trywrlock*( ).

NAME

pthread_rwlockattr_destroy, pthread_rwlockattr_init — destroy and initialize read-write lock attributes object

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

DESCRIPTION

The *pthread_rwlockattr_destroy*( ) function shall destroy a read-write lock attributes object. A destroyed *attr* attributes object can be reinitialized using *pthread_rwlockattr_init*( ); the results of otherwise referencing the object after it has been destroyed are undefined. An implementation may cause *pthread_rwlockattr_destroy*( ) to set the object referenced by *attr* to an invalid value.

The *pthread_rwlockattr_init*( ) function shall initialize a read-write lock attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if *pthread_rwlockattr_init*( ) is called specifying an already initialized *attr* attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) shall not affect any previously initialized read-write locks.

RETURN VALUE

If successful, the *pthread_rwlockattr_destroy*( ) and *pthread_rwlockattr_init*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlockattr_destroy*( ) function may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_rwlockattr_init*( ) function shall fail if:

[ENOMEM]	Insufficient memory exists to initialize the read-write lock attributes object.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_init( ), *pthread_rwlockattr_getpshared*( ), *pthread_rwlockattr_setpshared*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The SEE ALSO section is updated.

NAME

pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared — get and set process-shared attribute of read-write lock attributes object

SYNOPSIS

THR TSH

```
#include <pthread.h>

int pthread_rwlockattr_getpshared(
    const pthread_rwlockattr_t *restrict attr,
    int *restrict pshared);
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```

DESCRIPTION

The *pthread_rwlockattr_getpshared*( ) function shall obtain the value of the *process-shared* attribute from the initialized attributes object referenced by *attr*. The *pthread_rwlockattr_setpshared*( ) function shall set the *process-shared* attribute in an initialized attributes object referenced by *attr*.

The *process-shared* attribute shall be set to PTHREAD_PROCESS_SHARED to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the *process-shared* attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock shall only be operated upon by threads created within the same process as the thread that initialized the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behavior is undefined. The default value of the *process-shared* attribute shall be PTHREAD_PROCESS_PRIVATE.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

RETURN VALUE

Upon successful completion, the *pthread_rwlockattr_getpshared*( ) shall return zero and store the value of the *process-shared* attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

If successful, the *pthread_rwlockattr_setpshared*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_rwlockattr_getpshared*( ) and *pthread_rwlockattr_setpshared*( ) functions may fail if:

[EINVAL]	The value specified by *attr* is invalid.

The *pthread_rwlockattr_setpshared*( ) function may fail if:

[EINVAL]	The new value specified for the attribute is outside the range of legal values for that attribute.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_rwlock_init( ), *pthread_rwlockattr_destroy*( ), *pthread_rwlockattr_init*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5.

Issue 6

The following changes are made for alignment with IEEE Std 1003.1j-2000:

- The margin code in the SYNOPSIS is changed to THR TSH to indicate that the functionality is now part of the Threads option (previously it was part of the Read-Write Locks option in IEEE Std 1003.1j-2000 and also part of the XSI extension).
- The DESCRIPTION notes that additional attributes are implementation-defined.
- The SEE ALSO section is updated.

The **restrict** keyword is added to the *pthread_rwlockattr_getpshared*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_rwlockattr_init — initialize read-write lock attributes object

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

DESCRIPTION

Refer to *pthread_rwlockattr_destroy*( ).

NAME

pthread_rwlockattr_setpshared — set process-shared attribute of read-write lock attributes object

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
    int pshared);
```

DESCRIPTION

Refer to *pthread_rwlockattr_getpshared*( ).

NAME

pthread_self — get calling thread's ID

SYNOPSIS

THR
```
#include <pthread.h>

pthread_t pthread_self(void);
```

DESCRIPTION

The *pthread_self*( ) function shall return the thread ID of the calling thread.

RETURN VALUE

Refer to the DESCRIPTION.

ERRORS

No errors are defined.

The *pthread_self*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pthread_self*( ) function provides a capability similar to the *getpid*( ) function for processes and the rationale is the same: the creation call does not provide the thread ID to the created thread.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_create( ), *pthread_equal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_self*( ) function is marked as part of the Threads option.

NAME

pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel — set cancelability state

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

DESCRIPTION

The *pthread_setcancelstate*( ) function shall atomically both set the calling thread's cancelability state to the indicated *state* and return the previous cancelability state at the location referenced by *oldstate*. Legal values for *state* are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

The *pthread_setcanceltype*( ) function shall atomically both set the calling thread's cancelability type to the indicated *type* and return the previous cancelability type at the location referenced by *oldtype*. Legal values for *type* are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.

The cancelability state and type of any newly created threads, including the thread in which *main*( ) was first invoked, shall be PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

The *pthread_testcancel*( ) function shall create a cancelation point in the calling thread. The *pthread_testcancel*( ) function shall have no effect if cancelability is disabled.

RETURN VALUE

If successful, the *pthread_setcancelstate*( ) and *pthread_setcanceltype*( ) functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_setcancelstate*( ) function may fail if:

[EINVAL]	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

The *pthread_setcanceltype*( ) function may fail if:

[EINVAL]	The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pthread_setcancelstate*( ) and *pthread_setcanceltype*( ) functions control the points at which a thread may be asynchronously canceled. For cancelation control to be usable in modular fashion, some rules need to be followed.

An object can be considered to be a generalization of a procedure. It is a set of procedures and global variables written as a unit and called by clients not known by the object. Objects may depend on other objects.

First, cancelability should only be disabled on entry to an object, never explicitly enabled. On exit from an object, the cancelability state should always be restored to its value on entry to the object.

This follows from a modularity argument: if the client of an object (or the client of an object that uses that object) has disabled cancelability, it is because the client does not want to be concerned about cleaning up if the thread is canceled while executing some sequence of actions. If an object is called in such a state and it enables cancelability and a cancelation request is pending for that thread, then the thread is canceled, contrary to the wish of the client that disabled.

Second, the cancelability type may be explicitly set to either *deferred* or *asynchronous* upon entry to an object. But as with the cancelability state, on exit from an object the cancelability type should always be restored to its value on entry to the object.

Finally, only functions that are cancel-safe may be called from a thread that is asynchronously cancelable.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_cancel( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_setcancelstate*( ), *pthread_setcanceltype*( ), and *pthread_testcancel*( ) functions are marked as part of the Threads option.

NAME

pthread_setconcurrency — set level of concurrency

SYNOPSIS

XSI

```
#include <pthread.h>

int pthread_setconcurrency(int new_level);

```

DESCRIPTION

Refer to *pthread_getconcurrency*( ).

NAME

pthread_setschedparam — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

THR TPS

```
#include <pthread.h>

int pthread_setschedparam(pthread_t thread, int policy,
    const struct sched_param *param);
```

DESCRIPTION

Refer to *pthread_getschedparam*( ).

NAME

pthread_setschedprio — dynamic thread scheduling parameters access (**REALTIME THREADS**)

SYNOPSIS

THR TPS
```
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```

DESCRIPTION

The *pthread_setschedprio*( ) function shall set the scheduling priority for the thread whose thread ID is given by *thread* to the value given by *prio*. See **Scheduling Policies** (on page 494) for a description on how this function call affects the ordering of the thread in the thread list for its new priority.

If the *pthread_setschedprio*( ) function fails, the scheduling priority of the target thread shall not be changed.

RETURN VALUE

If successful, the *pthread_setschedprio*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_setschedprio*( ) function may fail if:

[EINVAL]	The value of *prio* is invalid for the scheduling policy of the specified thread.
[ENOTSUP]	An attempt was made to set the priority to an unsupported value.
[EPERM]	The caller does not have the appropriate permission to set the scheduling policy of the specified thread.
[EPERM]	The implementation does not allow the application to modify the priority to the value specified.
[ESRCH]	The value specified by *thread* does not refer to an existing thread.

The *pthread_setschedprio*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *pthread_setschedprio*( ) function provides a way for an application to temporarily raise its priority and then lower it again, without having the undesired side effect of yielding to other threads of the same priority. This is necessary if the application is to implement its own strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This capability is especially important if the implementation does not support the Thread Priority Protection or Thread Priority Inheritance options, but even if those options are supported it is needed if the application is to bound priority inheritance for other resources, such as semaphores.

The standard developers considered that while it might be preferable conceptually to solve this problem by modifying the specification of *pthread_setschedparam*( ), it was too late to make such a change, as there may be implementations that would need to be changed. Therefore, this new function was introduced.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_getschedparam( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Included as a response to IEEE PASC Interpretation 1003.1 #96.

NAME

pthread_setspecific — thread-specific data management

SYNOPSIS

THR

```
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, const void *value);
```

DESCRIPTION

Refer to *pthread_getspecific*( ).

NAME

pthread_sigmask, sigprocmask — examine and change blocked signals

SYNOPSIS

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
int sigprocmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```

DESCRIPTION

THR The *pthread_sigmask*( ) function shall examine or change (or both) the calling thread's signal mask, regardless of the number of threads in the process. The function shall be equivalent to *sigprocmask*( ), without the restriction that the call be made in a single-threaded process.

In a single-threaded process, the *sigprocmask*( ) function shall examine or change (or both) the signal mask of the calling thread.

If the argument *set* is not a null pointer, it points to a set of signals to be used to change the currently blocked set.

The argument *how* indicates the way in which the set is changed, and the application shall ensure it consists of one of the following values:

SIG_BLOCK	The resulting set shall be the union of the current set and the signal set pointed to by *set*.
SIG_SETMASK	The resulting set shall be the signal set pointed to by *set*.
SIG_UNBLOCK	The resulting set shall be the intersection of the current set and the complement of the signal set pointed to by *set*.

If the argument *oset* is not a null pointer, the previous mask shall be stored in the location pointed to by *oset*. If *set* is a null pointer, the value of the argument *how* is not significant and the process' signal mask shall be unchanged; thus the call can be used to enquire about currently blocked signals.

If there are any pending unblocked signals after the call to *sigprocmask*( ), at least one of those signals shall be delivered before the call to *sigprocmask*( ) returns.

It is not possible to block those signals which cannot be ignored. This shall be enforced by the system without causing an error to be indicated.

If any of the SIGFPE, SIGILL, SIGSEGV, or SIGBUS signals are generated while they are blocked, the result is undefined, unless the signal was generated by the *kill*( ) function, the *sigqueue*( ) function, or the *raise*( ) function.

If *sigprocmask*( ) fails, the thread's signal mask shall not be changed.

The use of the *sigprocmask*( ) function is unspecified in a multi-threaded process.

RETURN VALUE

THR Upon successful completion *pthread_sigmask*( ) shall return 0; otherwise, it shall return the corresponding error number.

Upon successful completion, *sigprocmask*( ) shall return 0; otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and the process' signal mask shall be unchanged.

ERRORS

THR The *pthread_sigmask*( ) and *sigprocmask*( ) functions shall fail if:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

THR The *pthread_sigmask*( ) function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

When a process' signal mask is changed in a signal-catching function that is installed by *sigaction*( ), the restoration of the signal mask on return from the signal-catching function overrides that change (see *sigaction*( )). If the signal-catching function was installed with *signal*( ), it is unspecified whether this occurs.

See *kill*( ) for a discussion of the requirement on delivery of signals.

FUTURE DIRECTIONS

None.

SEE ALSO

sigaction( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ), *sigqueue*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

The *pthread_sigmask*( ) function is added for alignment with the POSIX Threads Extension.

Issue 6

The *pthread_sigmask*( ) function is marked as part of the Threads option.

The SYNOPSIS for *sigprocmask*( ) is marked as a CX extension to note that the presence of this function in the **<signal.h>** header is an extension to the ISO C standard. |

The following changes are made for alignment with the ISO POSIX-1: 1996 standard: |

- The DESCRIPTION is updated to explicitly state the functions which may generate the signal.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The **restrict** keyword is added to the *pthread_sigmask*( ) and *sigprocmask*( ) prototypes for alignment with the ISO / IEC 9899: 1999 standard.

NAME

pthread_spin_destroy, pthread_spin_init — destroy or initialize a spin lock object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR SPI
```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

DESCRIPTION

The *pthread_spin_destroy*( ) function shall destroy the spin lock referenced by *lock* and release any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to *pthread_spin_init*( ). The results are undefined if *pthread_spin_destroy*( ) is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

The *pthread_spin_init*( ) function shall allocate any resources required to use the spin lock referenced by *lock* and initialize the lock to an unlocked state.

TSH If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

The results are undefined if *pthread_spin_init*( ) is called specifying an already initialized spin lock. The results are undefined if a spin lock is used without first being initialized.

If the *pthread_spin_init*( ) function fails, the lock is not initialized and the contents of *lock* are undefined.

Only the object referenced by *lock* may be used for performing synchronization.

The result of referring to copies of that object in calls to *pthread_spin_destroy*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), or *pthread_spin_unlock*( ) is undefined.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EBUSY]	The implementation has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a *pthread_spin_lock*( ) call) by another thread.
[EINVAL]	The value specified by *lock* is invalid.

The *pthread_spin_init*( ) function shall fail if:

[EAGAIN]	The system lacks the necessary resources to initialize another spin lock.

[ENOMEM] Insufficient memory exists to initialize the lock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_spin_destroy*( ) and *pthread_spin_init*( ) functions are part of the Spin Locks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_lock( ), *pthread_spin_trylock*( ), *pthread_spin_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<<pthread.h>>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_spin_init — initialize a spin lock object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR SPI

```
#include <pthread.h>

int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

DESCRIPTION

Refer to *pthread_spin_destroy*( ).

NAME

pthread_spin_lock, pthread_spin_trylock — lock a spin lock object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR SPI

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

DESCRIPTION

The *pthread_spin_lock*( ) function shall lock the spin lock referenced by *lock*. The calling thread shall acquire the lock if it is not held by another thread. Otherwise, the thread shall spin (that is, shall not return from the *pthread_spin_lock*( ) call) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made. The *pthread_spin_trylock*( ) function shall lock the spin lock referenced by *lock* if it is not held by any thread. Otherwise, the function shall fail.

The results are undefined if any of these functions is called with an uninitialized spin lock.

RETURN VALUE

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL] The value specified by *lock* does not refer to an initialized spin lock object.

The *pthread_spin_lock*( ) function may fail if:

[EDEADLK] The calling thread already holds the lock.

The *pthread_spin_trylock*( ) function shall fail if:

[EBUSY] A thread currently holds the lock.

These functions shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

Applications using this function may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *pthread_spin_lock*( ) and *pthread_spin_trylock*( ) functions are part of the Spin Locks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_init( ), *pthread_spin_destroy*( ), *pthread_spin_unlock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_spin_trylock — lock a spin lock object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR SPI

```
#include <pthread.h>

int pthread_spin_trylock(pthread_spinlock_t *lock);

```

DESCRIPTION

Refer to *pthread_spin_lock*( ).

NAME

pthread_spin_unlock — unlock a spin lock object (**ADVANCED REALTIME THREADS**)

SYNOPSIS

THR SPI

```
#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);

```

DESCRIPTION

The *pthread_spin_unlock*( ) function shall release the spin lock referenced by *lock* which was locked via the *pthread_spin_lock*( ) or *pthread_spin_trylock*( ) functions. The results are undefined if the lock is not held by the calling thread. If there are threads spinning on the lock when *pthread_spin_unlock*( ) is called, the lock becomes available and an unspecified spinning thread shall acquire the lock.

The results are undefined if this function is called with an uninitialized thread spin lock.

RETURN VALUE

Upon successful completion, the *pthread_spin_unlock*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *pthread_spin_unlock*( ) function may fail if:

[EINVAL]	An invalid argument was specified.
[EPERM]	The calling thread does not hold the lock.

This function shall not return an error code of [EINTR].

EXAMPLES

None.

APPLICATION USAGE

The *pthread_spin_unlock*( ) function is part of the Spin Locks option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pthread_spin_init( ), *pthread_spin_destroy*( ), *pthread_spin_lock*( ), *pthread_spin_trylock*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<pthread.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1j-2000.

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

pthread_testcancel — set cancelability state

SYNOPSIS

THR
```
#include <pthread.h>

void pthread_testcancel(void);
```

DESCRIPTION

Refer to *pthread_setcancelstate*( ).

NAME

ptsname — get name of the slave pseudo-terminal device

SYNOPSIS

XSI
```
#include <stdlib.h>

char *ptsname(int fildes);
```

DESCRIPTION

The *ptsname*( ) function shall return the name of the slave pseudo-terminal device associated with a master pseudo-terminal device. The *fildes* argument is a file descriptor that refers to the master device. The *ptsname*( ) function shall return a pointer to a string containing the pathname of the corresponding slave device. |

The *ptsname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *ptsname*( ) shall return a pointer to a string which is the name of the pseudo-terminal slave device. Upon failure, *ptsname*( ) shall return a null pointer. This could occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file system.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The value returned may point to a static data area that is overwritten by each call to *ptsname*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

grantpt( ), *open*( ), *ttyname*( ), *unlockpt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

putc — put byte on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *putc*( ) function shall be equivalent to *fputc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

RETURN VALUE

Refer to *fputc*( ).

ERRORS

Refer to *fputc*( ).

EXAMPLES

None.

APPLICATION USAGE

Since it may be implemented as a macro, *putc*( ) may treat a *stream* argument with side effects incorrectly. In particular, *putc*(*c*,**f*++) does not necessarily work correctly. Therefore, use of this function is not recommended in such situations; *fputc*( ) should be used instead. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

putc_unlocked — stdio with explicit client locking

SYNOPSIS

TSF

```
#include <stdio.h>

int putc_unlocked(int c, FILE *stream);
```

DESCRIPTION

Refer to *getc_unlocked*( ).

NAME

putchar — put byte on stdout stream

SYNOPSIS

```
#include <stdio.h>

int putchar(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The function call *putchar*(*c*) shall be equivalent to *putc*(*c*,*stdout*).

RETURN VALUE

Refer to *fputc*( ).

ERRORS

Refer to *fputc*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

putc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

putchar_unlocked — stdio with explicit client locking

SYNOPSIS

TSF

```
#include <stdio.h>

int putchar_unlocked(int c);
```

DESCRIPTION

Refer to *getc_unlocked*( ).

NAME

putenv — change or add a value to environment

SYNOPSIS

XSI

```
#include <stdlib.h>

int putenv(char *string);
```

DESCRIPTION

The *putenv*( ) function shall use the *string* argument to set environment variable values. The *string* argument should point to a string of the form "*name*=*value*. The *putenv*( ) function shall make the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* shall become part of the environment, so altering the string shall change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to *putenv*( ).

The *putenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, *putenv*( ) shall return 0; otherwise, it shall return a non-zero value and set *errno* to indicate the error.

ERRORS

The *putenv*( ) function may fail if:

[ENOMEM]	Insufficient memory was available.

EXAMPLES

Changing the Value of an Environment Variable

The following example changes the value of the *HOME* environment variable to the value **/usr/home**.

```
#include <stdlib.h>
...
static char *var = "HOME=/usr/home";
int ret;

ret = putenv(var);
```

APPLICATION USAGE

The *putenv*( ) function manipulates the environment pointed to by *environ*, and can be used in conjunction with *getenv*( ).

This routine may use *malloc*( ) to enlarge the environment.

A potential error is to call *putenv*( ) with an automatic variable as the argument, then return from the calling function while *string* is still part of the environment.

The *setenv*( ) function is preferred over this function.

RATIONALE

The standard developers noted that *putenv*( ) is the only function available to add to the environment without permitting memory leaks.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getenv*( ), *malloc*( ), *setenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The type of the argument to this function is changed from **const char *** to **char ***. This was indicated as a FUTURE DIRECTION in previous issues.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

NAME

putmsg, putpmsg — send a message on a STREAM (**STREAMS**)

SYNOPSIS

XSR
```
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int flags);
int putpmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION

The *putmsg*( ) function shall create a message from a process buffer(s) and send the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts are distinguished by placement in separate buffers, as described below. The semantics of each part are defined by the STREAMS module that receives the message.

The *putpmsg*( ) function is equivalent o *putmsg*( ), except that the process can send messages in different priority bands. Except where noted, all requirements on *putmsg*( ) also pertain to *putpmsg*( ).

The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure.

The *ctlptr* argument points to the structure describing the control part, if any, to be included in the message. The *buf* member in the **strbuf** structure points to the buffer where the control information resides, and the *len* member indicates the number of bytes to be sent. The *maxlen* member is not used by *putmsg*( ). In a similar manner, the argument *dataptr* specifies the data, if any, to be included in the message. The *flags* argument indicates what type of message should be sent and is described further below.

To send the data part of a message, the application shall ensure that *dataptr* is not a null pointer and the *len* member of *dataptr* is 0 or greater. To send the control part of a message, the application shall ensure that the corresponding values are set for *ctlptr*. No data (control) part shall be sent if either *dataptr*(*ctlptr*) is a null pointer or the *len* member of *dataptr*(*ctlptr*) is set to −1.

For *putmsg*( ), if a control part is specified and *flags* is set to RS_HIPRI, a high priority message shall be sent. If no control part is specified, and *flags* is set to RS_HIPRI, *putmsg*( ) shall fail and set *errno* to [EINVAL]. If *flags* is set to 0, a normal message (priority band equal to 0) shall be sent. If a control part and data part are not specified and *flags* is set to 0, no message shall be sent and 0 shall be returned. |

For *putpmsg*( ), the flags are different. The *flags* argument is a bitmask with the following mutually-exclusive flags defined: MSG_HIPRI and MSG_BAND. If *flags* is set to 0, *putpmsg*( ) shall fail and set *errno* to [EINVAL]. If a control part is specified and *flags* is set to MSG_HIPRI and *band* is set to 0, a high-priority message shall be sent. If *flags* is set to MSG_HIPRI and either no control part is specified or *band* is set to a non-zero value, *putpmsg*( ) shall fail and set *errno* to [EINVAL]. If *flags* is set to MSG_BAND, then a message shall be sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to MSG_BAND, no message shall be sent and 0 shall be returned. |

The *putmsg*( ) function shall block if the STREAM write queue is full due to internal flow control conditions, with the following exceptions: |

- For high-priority messages, *putmsg*( ) shall not block on this condition and continues processing the message. |

- For other messages, *putmsg*( ) shall not block but shall fail when the write queue is full and O_NONBLOCK is set.

The *putmsg*( ) function shall also block, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message shall be sent.

RETURN VALUE

Upon successful completion, *putmsg*( ) and *putpmsg*( ) shall return 0; otherwise, they shall return −1 and set *errno* to indicate the error.

ERRORS

The *putmsg*( ) and *putpmsg*( ) functions shall fail if:

[EAGAIN]	A non-priority message was specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions; or buffers could not be allocated for the message that was to be created.
[EBADF]	*fildes* is not a valid file descriptor open for writing.
[EINTR]	A signal was caught during *putmsg*( ).
[EINVAL]	An undefined value is specified in *flags*, or *flags* is set to RS_HIPRI or MSG_HIPRI and no control part is supplied, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer, or *flags* is set to MSG_HIPRI and *band* is non-zero (for *putpmsg*( ) only).
[ENOSR]	Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.
[ENOSTR]	A STREAM is not associated with *fildes*.
[ENXIO]	A hangup condition was generated downstream for the specified STREAM.
[EPIPE] or [EIO]	The *fildes* argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling thread.
[ERANGE]	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, *putmsg*( ) and *putpmsg*( ) shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *putmsg*( ) or *putpmsg*( ), but reflects the prior error.

EXAMPLES

Sending a High-Priority Message

The value of *fd* is assumed to refer to an open STREAMS file. This call to *putmsg*( ) does the following:

1. Creates a high-priority message with a control part and a data part, using the buffers pointed to by *ctrlbuf* and *databuf*, respectively.
2. Sends the message to the STREAMS file identified by *fd*.

```
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;

ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);

data.buf = databuf;
data.len = strlen(databuf);

ret = putmsg(fd, &ctrl, &data, MSG_HIPRI);
```

Using putpmsg( )

This example has the same effect as the previous example. In this example, however, the *putpmsg*( ) function creates and sends the message to the STREAMS file.

```
#include <stropts.h>
#include <string.h>
...
int fd;
char *ctrlbuf = "This is the control part";
char *databuf = "This is the data part";
struct strbuf ctrl;
struct strbuf data;
int ret;

ctrl.buf = ctrlbuf;
ctrl.len = strlen(ctrlbuf);

data.buf = databuf;
data.len = strlen(databuf);

ret = putpmsg(fd, &ctrl, &data, 0, MSG_HIPRI);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getmsg( ), *poll*( ), *read*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, Section 2.6 (on page 488)

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The following text is removed from the DESCRIPTION: ‘‘The STREAM head guarantees that the control part of a message generated by *putmsg*( ) is at least 64 bytes in length’’.

Issue 6

This function is marked as part of the XSI STREAMS Option Group.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

putpmsg — send a message on a STREAM (**STREAMS**)

SYNOPSIS

XSR

```
#include <stropts.h>

int putpmsg(int fildes, const struct strbuf *ctlptr,
    const struct strbuf *dataptr, int band, int flags);
```

DESCRIPTION

Refer to *putmsg*( ).

NAME

puts — put a string on standard output

SYNOPSIS

```
#include <stdio.h>

int puts(const char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *puts*( ) function shall write the string pointed to by *s*, followed by a <newline>, to the standard output stream *stdout*. The terminating null byte shall not be written.

CX The *st_ctime* and *st_mtime* fields of the file shall be marked for update between the successful execution of *puts*( ) and the next successful completion of a call to *fflush*( ) or *fclose*( ) on the same stream or a call to *exit*( ) or *abort*( ).

RETURN VALUE

Upon successful completion, *puts*( ) shall return a non-negative number. Otherwise, it shall return EOF, shall set an error indicator for the stream, and *errno* shall be set to indicate the error. CX

ERRORS

Refer to *fputc*( ).

EXAMPLES

Printing to Standard Output

The following example gets the current time, converts it to a string using *localtime*( ) and *asctime*( ), and prints it to standard output using *puts*( ). It then prints the number of minutes to an event for which it is waiting.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
    minutes_to_event);
...
```

APPLICATION USAGE

The *puts*( ) function appends a <newline>, while *fputs*( ) does not.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *fputs*( ), *putc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

pututxline — put an entry into user accounting database

SYNOPSIS

XSI

```
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *utmpx);
```

DESCRIPTION

Refer to *endutxent*( ).

NAME

putwc — put a wide character on a stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t wc, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *putwc*( ) function shall be equivalent to *fputwc*( ), except that if it is implemented as a macro it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

RETURN VALUE

Refer to *fputwc*( ).

ERRORS

Refer to *fputwc*( ).

EXAMPLES

None.

APPLICATION USAGE

Since it may be implemented as a macro, *putwc*( ) may treat a *stream* argument with side effects incorrectly. In particular, *putwc*(*wc*,**f*++) need not work correctly. Therefore, use of this function is not recommended; *fputwc*( ) should be used instead. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputwc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

The Optional Header (OH) marking is removed from **<stdio.h>**.

NAME

putwchar — put a wide character on stdout stream

SYNOPSIS

```
#include <wchar.h>

wint_t putwchar(wchar_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The function call *putwchar*(*wc*) shall be equivalent to *putwc*(*wc*,*stdout*).

RETURN VALUE

Refer to *fputwc*( ).

ERRORS

Refer to *fputwc*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fputwc( ), *putwc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of argument *wc* is changed from **wint_t** to **wchar_t**.

NAME

pwrite — write on a file

SYNOPSIS

```
#include <unistd.h>
```

XSI
```
ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
    off_t offset);
```

DESCRIPTION

Refer to *write*( ).

NAME

qsort — sort a table of data

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *qsort*( ) function shall sort an array of *nel* objects, the initial element of which is pointed to by *base*. The size of each object, in bytes, is specified by the *width* argument.

The contents of the array shall be sorted in ascending order according to a comparison function. The *compar* argument is a pointer to the comparison function, which is called with two arguments that point to the elements being compared. The application shall ensure that the function returns an integer less than, equal to, or greater than 0, if the first argument is considered respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

RETURN VALUE

The *qsort*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

raise — send a signal to the executing process

SYNOPSIS

```
#include <signal.h>

int raise(int sig);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *raise*( ) function shall send the signal *sig* to the executing thread or process. If a signal handler is called, the *raise*( ) function shall not return until after the signal handler does.

THR If the implementation supports the Threads option, the effect of the *raise*( ) function shall be equivalent to calling: |

```
pthread_kill(pthread_self(), sig);
```

CX Otherwise, the effect of the *raise*( ) function shall be equivalent to calling: |

```
kill(getpid(), sig);
```

RETURN VALUE

CX Upon successful completion, 0 shall be returned. Otherwise, a non-zero value shall be returned and *errno* shall be set to indicate the error.

ERRORS

The *raise*( ) function shall fail if:

CX [EINVAL] The value of the *sig* argument is an invalid signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The term ‘‘thread’’ is an extension to the ISO C standard.

FUTURE DIRECTIONS

None.

SEE ALSO

kill( ), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the requirement to set *errno* on error is added.
- The [EINVAL] error condition is added.

NAME

rand, rand_r, srand — pseudo-random number generator

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
int rand_r(unsigned *seed);
void srand(unsigned seed);
```

TSF (rand_r)

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

XSI The *rand*( ) function shall compute a sequence of pseudo-random integers in the range 0 to {RAND_MAX} with a period of at least 2^{32}.

CX The *rand*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *rand_r*( ) function shall compute a sequence of pseudo-random integers in the range 0 to {RAND_MAX}. (The value of the {RAND_MAX} macro shall be at least 32 767.)

If *rand_r*( ) is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to *rand_r*( ), the same sequence shall be generated.

The *srand*( ) function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to *rand*( ). If *srand*( ) is then called with the same seed value, the sequence of pseudo-random numbers shall be repeated. If *rand*( ) is called before any calls to *srand*( ) are made, the same sequence shall be generated as when *srand*( ) is first called with a seed value of 1.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *rand*( ) or *srand*( ).

RETURN VALUE

The *rand*( ) function shall return the next pseudo-random number in the sequence.

TSF The *rand_r*( ) function shall return a pseudo-random integer.

The *srand*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Generating a Pseudo-Random Number Sequence

The following example demonstrates how to generate a sequence of pseudo-random numbers.

```
#include <stdio.h>
#include <stdlib.h>
...
    long count, i;
    char *keystr;
    int elementlen, len;
    char c;
```

```
...
/* Initial random number generator. */
    srand(1);

    /* Create keys using only lower case characters */
    len = 0;
    for (i=0; i<count; i++) {
        while (len < elementlen) {
            c = (char) (rand() % 128);
            if (islower(c))
                keystr[len++] = c;
        }

        keystr[len] = '\0';
        printf("%s Element%0*ld\n", keystr, elementlen, i);
        len = 0;
    }
```

Generating the Same Sequence on Different Machines

The following code defines a pair of functions that could be incorporated into applications wishing to ensure that the same sequence of numbers is generated across different machines.

```
static unsigned long next = 1;
int myrand(void)  /* RAND_MAX assumed to be 32767. */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}
```

APPLICATION USAGE

The *drand48*( ) function provides a much more elaborate random number generator.

RATIONALE

The ISO C standard *rand*( ) and *srand*( ) functions allow per-process pseudo-random streams shared by all threads. Those two functions need not change, but there has to be mutual-exclusion that prevents interference between two threads concurrently accessing the random number generator.

With regard to *rand*( ), there are two different behaviors that may be wanted in a multi-threaded program:

1. A single per-process sequence of pseudo-random numbers that is shared by all threads that call *rand*( )

2. A different sequence of pseudo-random numbers for each thread that calls *rand*( )

This is provided by the modified thread-safe function based on whether the seed value is global to the entire process or local to each thread.

This does not address the known deficiencies of the *rand*( ) function implementations, which have been approached by maintaining more state. In effect, this specifies new thread-safe forms of a deficient function. |

FUTURE DIRECTIONS

None.

SEE ALSO

drand48( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The *rand_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *rand*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *rand_r*( ) function is marked as part of the Thread-Safe Functions option.

NAME

random — generate pseudo-random number

SYNOPSIS

XSI

```
#include <stdlib.h>

long random(void);
```

DESCRIPTION

Refer to *initstate*( ).

NAME

pread, read — read from a file

SYNOPSIS

```
#include <unistd.h>

XSI ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
ssize_t read(int fildes, void *buf, size_t nbyte);
```

DESCRIPTION

The *read*( ) function shall attempt to read *nbyte* bytes from the file associated with the open file descriptor, *fildes*, into the buffer pointed to by *buf*. The behavior of multiple concurrent reads on the same pipe, FIFO, or terminal device is unspecified.

Before any action described below is taken, and if *nbyte* is zero, the *read*( ) function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the *read*( ) function shall return zero and have no other results.

On files that support seeking (for example, a regular file), the *read*( ) shall start at a position in the file given by the file offset associated with *fildes*. The file offset shall be incremented by the number of bytes actually read.

Files that do not support seeking—for example, terminals—always read from the current position. The value of a file offset associated with such a file is undefined.

No data transfer shall occur past the current end-of-file. If the starting position is at or after the end-of-file, 0 shall be returned. If the file refers to a device special file, the result of subsequent *read*( ) requests is implementation-defined.

If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

When attempting to read from an empty pipe or FIFO:

- If no process has the pipe open for writing, *read*( ) shall return 0 to indicate end-of-file.
- If some process has the pipe open for writing and O_NONBLOCK is set, *read*( ) shall return −1 and set *errno* to [EAGAIN].
- If some process has the pipe open for writing and O_NONBLOCK is clear, *read*( ) shall block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

- If O_NONBLOCK is set, *read*( ) shall return −1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is clear, *read*( ) shall block the calling thread until some data becomes available.
- The use of the O_NONBLOCK flag has no effect if there is some data available.

The *read*( ) function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, *read*( ) shall return bytes with value 0. For example, *lseek*( ) allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads in the gap between the previous end of data and the newly written data shall return bytes with value 0 until data is written into the gap.

Upon successful completion, where *nbyte* is greater than 0, *read*( ) shall mark for update the *st_atime* field of the file, and shall return the number of bytes read. This number shall never be greater than *nbyte*. The value returned may be less than *nbyte* if the number of bytes left in the file is less than *nbyte*, if the *read*( ) request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than *nbyte* bytes immediately available for reading. For example, a *read*( ) from a file associated with a terminal may return one typed line of data.

If a *read*( ) is interrupted by a signal before it reads any data, it shall return −1 with *errno* set to [EINTR].

If a *read*( ) is interrupted by a signal after it has successfully read some data, it shall return the number of bytes read.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildes*.

If *fildes* refers to a socket, *read*( ) shall be equivalent to *recv*( ) with no flags set.

SIO If the O_DSYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion. If the O_SYNC and O_RSYNC bits have been set, read I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

SHM If *fildes* refers to a shared memory object, the result of the *read*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *read*( ) function is unspecified.

XSR A *read*( ) from a STREAMS file can read data in three different modes: *byte-stream* mode, *message-nondiscard* mode, and *message-discard* mode. The default shall be byte-stream mode. This can be changed using the I_SRDOPT *ioctl*( ) request, and can be tested with the I_GRDOPT *ioctl*( ). In byte-stream mode, *read*( ) shall retrieve data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, *read*( ) shall retrieve data until as many bytes as were requested are transferred, or until a message boundary is reached. If *read*( ) does not retrieve all the data in a message, the remaining data shall be left on the STREAM, and can be retrieved by the next *read*( ) call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the *read*( ) returns shall be discarded, and shall not be available for a subsequent *read*( ), *getmsg*( ), or *getpmsg*( ) call.

How *read*( ) handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, *read*( ) shall accept data until it has read *nbyte* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The *read*( ) function shall then return the number of bytes read, and place the zero-byte message back on the STREAM to be retrieved by the next *read*( ), *getmsg*( ), or *getpmsg*( ). In message-nondiscard mode or message-discard mode, a zero-byte message shall return 0 and the message shall be removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message shall be removed from the STREAM and 0 shall be returned, regardless of the read mode.

A *read*( ) from a STREAMS file shall return the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMs are in control-normal mode, in which a *read*( ) from a STREAMS file can only process messages that contain a data part but do not contain a control part. The *read*( ) shall fail if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the I_SRDOPT *ioctl*( ) command. In control-data mode, *read*( ) shall convert any control part to data and pass it to the application before passing any data part originally present in the same message. In control-discard mode, *read*( ) shall discard message control parts but return to the process any data part in the message.

In addition, *read*( ) shall fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of *errno* shall not reflect the result of *read*( ), but reflects the prior error. If a hangup occurs on the STREAM being read, *read*( ) shall continue to operate normally until the STREAM head read queue is empty. Thereafter, it shall return 0.

XSI The *pread*( ) function shall be equivalent to *read*( ), except that it shall read from a given position in the file without changing the file pointer. The first three arguments to *pread*( ) are the same as *read*( ) with the addition of a fourth argument offset for the desired position inside the file. An attempt to perform a *pread*( ) on a file that is incapable of seeking shall result in an error.

RETURN VALUE

XSI Upon successful completion, *read*( ) and *pread*( ) shall return a non-negative integer indicating the number of bytes actually read. Otherwise, the functions shall return −1 and set *errno* to indicate the error.

ERRORS

XSI The *read*( ) and *pread*( ) functions shall fail if:

	[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the process would be delayed.
	[EBADF]	The *fildes* argument is not a valid file descriptor open for reading.
XSR	[EBADMSG]	The file is a STREAM file that is set to control-normal mode and the message waiting to be read includes a control part.
	[EINTR]	The read operation was terminated due to the receipt of a signal, and no data was transferred.
XSR	[EINVAL]	The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.
	[EIO]	The process is a member of a background process attempting to read from its controlling terminal, the process is ignoring or blocking the SIGTTIN signal, or the process group is orphaned. This error may also be generated for implementation-defined reasons.
XSI	[EISDIR]	The *fildes* argument refers to a directory and the implementation does not allow the directory to be read using *read*( ) or *pread*( ). The *readdir*( ) function should be used instead.
	[EOVERFLOW]	The file is a regular file, *nbyte* is greater than 0, the starting position is before the end-of-file, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.

The *read*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The file descriptor is for a socket, is marked O_NONBLOCK, and no data is waiting to be received.
[ECONNRESET]	A read was attempted on a socket and the connection was forcibly closed by its peer.
[ENOTCONN]	A read was attempted on a socket that is not connected.
[ETIMEDOUT]	A read was attempted on a socket and a transmission timeout occurred.

XSI The *read*( ) and *pread*( ) functions may fail if:

[EIO]	A physical I/O error has occurred.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.
[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

The *pread*( ) function shall fail, and the file pointer shall remain unchanged, if:

XSI	[EINVAL]	The *offset* argument is invalid. The value is negative.
XSI	[EOVERFLOW]	The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the file.
XSI	[ENXIO]	A request was outside the capabilities of the device.
XSI	[ESPIPE]	*fildes* is associated with a pipe or FIFO.

EXAMPLES

Reading Data into a Buffer

The following example reads data from the file associated with the file descriptor *fd* into the buffer pointed to by *buf*.

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

APPLICATION USAGE

None.

RATIONALE

This volume of IEEE Std 1003.1-200x does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

Note that a *read*( ) of zero bytes does not modify *st_atime*. A *read*( ) that requests more than zero bytes, but returns zero, shall modify *st_atime*.

Implementations are allowed, but not required, to perform error checking for *read*( ) requests of zero bytes.

Input and Output

The use of I/O with large byte counts has always presented problems. Ideas such as *lread*( ) and *lwrite*( ) (using and returning **long**s) were considered at one time. The current solution is to use abstract types on the ISO C standard function to *read*( ) and *write*( ). The abstract types can be declared so that existing functions work, but can also be declared so that larger types can be represented in future implementations. It is presumed that whatever constraints limit the maximum range of **size_t** also limit portable I/O requests to the same range. This volume of IEEE Std 1003.1-200x also limits the range further by requiring that the byte count be limited so that a signed return value remains meaningful. Since the return type is also a (signed) abstract type, the byte count can be defined by the implementation to be larger than an **int** can hold.

The standard developers considered adding atomicity requirements to a pipe or FIFO, but recognized that due to the nature of pipes and FIFOs there could be no guarantee of atomicity of reads of {PIPE_BUF} or any other size that would be an aid to applications portability.

This volume of IEEE Std 1003.1-200x requires that no action be taken for *read*( ) or *write*( ) when *nbyte* is zero. This is not intended to take precedence over detection of errors (such as invalid buffer pointers or file descriptors). This is consistent with the rest of this volume of IEEE Std 1003.1-200x, but the phrasing here could be misread to require detection of the zero case before any other errors. A value of zero is to be considered a correct value, for which the semantics are a no-op.

I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. It is a known attribute of terminals that this is not honored, and terminals are explicitly (and implicitly permanently) excepted, making the behavior unspecified. The behavior for other device types is also left unspecified, but the wording is intended to imply that future standards might choose to specify atomicity (or not).

There were recommendations to add format parameters to *read*( ) and *write*( ) in order to handle networked transfers among heterogeneous file system and base hardware types. Such a facility may be required for support by the OSI presentation of layer services. However, it was determined that this should correspond with similar C-language facilities, and that is beyond the scope of this volume of IEEE Std 1003.1-200x. The concept was suggested to the developers of the ISO C standard for their consideration as a possible area for future work.

In 4.3 BSD, a *read*( ) or *write*( ) that is interrupted by a signal before transferring any data does not by default return an [EINTR] error, but is restarted. In 4.2 BSD, 4.3 BSD, and the Eighth Edition, there is an additional function, *select*( ), whose purpose is to pause until specified activity (data to read, space to write, and so on) is detected on specified file descriptors. It is common in applications written for those systems for *select*( ) to be used before *read*( ) in situations (such as keyboard input) where interruption of I/O due to a signal is desired.

The issue of which files or file types are interruptible is considered an implementation design issue. This is often affected primarily by hardware and reliability issues.

There are no references to actions taken following an ‘‘unrecoverable error’’. It is considered beyond the scope of this volume of IEEE Std 1003.1-200x to describe what happens in the case of hardware errors.

Previous versions of IEEE Std 1003.1-200x allowed two very different behaviors with regard to the handling of interrupts. In order to minimize the resulting confusion, it was decided that IEEE Std 1003.1-200x should support only one of these behaviors. Historical practice on AT&T-derived systems was to have *read*( ) and *write*( ) return −1 and set *errno* to [EINTR] when interrupted after some, but not all, of the data requested had been transferred. However, the U.S. Department of Commerce FIPS 151-1 and FIPS 151-2 require the historical BSD behavior, in which *read*( ) and *write*( ) return the number of bytes actually transferred before the interrupt. If –1 is returned when any data is transferred, it is difficult to recover from the error on a seekable device and impossible on a non-seekable device. Most new implementations support this behavior. The behavior required by IEEE Std 1003.1-200x is to return the number of bytes transferred.

IEEE Std 1003.1-200x does not specify when an implementation that buffers *read*( )s actually moves the data into the user-supplied buffer, so an implementation may chose to do this at the latest possible moment. Therefore, an interrupt arriving earlier may not cause *read*( ) to return a partial byte count, but rather to return –1 and set *errno* to [EINTR].

Consideration was also given to combining the two previous options, and setting *errno* to [EINTR] while returning a short count. However, not only is there no existing practice that implements this, it is also contradictory to the idea that when *errno* is set, the function responsible shall return –1.

FUTURE DIRECTIONS

None.

SEE ALSO

fcntl( ), *ioctl*( ), *lseek*( ), *open*( ), *pipe*( ), *readv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

The *pread*( ) function is added.

Issue 6

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now states that if *read*( ) is interrupted by a signal after it has successfully read some data, it returns the number of bytes read. In Issue 3, it was optional whether *read*( ) returned the number of bytes read, or whether it returned –1 with *errno* set to [EINTR]. This is a FIPS requirement.
- In the DESCRIPTION, text is added to indicate that for regular files, no data transfer occurs past the offset maximum established in the open file description associated with *fildes*. This change is to support large files.
- The [EOVERFLOW] mandatory error condition is added.
- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *read*( ) results are unspecified for typed memory objects.

New RATIONALE is added to explain the atomicity requirements for input and output operations.

The following error conditions are added for operations on sockets: [EAGAIN], [ECONNRESET], [ENOTCONN], and [ETIMEDOUT].

The [EIO] error is changed to ‘‘may fail’’.

The following error conditions are added for operations on sockets: [ENOBUFS] and [ENOMEM]. |

The *readv*( ) function is split out into a separate reference page. |

NAME

readdir, readdir_r — read directory

SYNOPSIS

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
int readdir_r(DIR *restrict dirp, struct dirent *restrict entry,
    struct dirent **restrict result);
```

DESCRIPTION

The type **DIR**, which is defined in the **<dirent.h>** header, represents a *directory stream*, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of *readdir*( ).

The *readdir*( ) function shall return a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument *dirp*, and position the directory stream at the next entry. It shall return a null pointer upon reaching the end of the directory stream. The structure **dirent** defined in the **<dirent.h>** header describes a directory entry. |

The *readdir*( ) function shall not return directory entries containing empty names. If entries for dot or dot-dot exist, one entry shall be returned for dot and one entry shall be returned for dot-dot; otherwise, they shall not be returned.

The pointer returned by *readdir*( ) points to data which may be overwritten by another call to *readdir*( ) on the same directory stream. This data is not overwritten by another call to *readdir*( ) on a different directory stream.

If a file is removed from or added to the directory after the most recent call to *opendir*( ) or *rewinddir*( ), whether a subsequent call to *readdir*( ) returns an entry for that file is unspecified.

The *readdir*( ) function may buffer several directory entries per actual read operation; *readdir*( ) shall mark for update the *st_atime* field of the directory each time the directory is actually read.

After a call to *fork*( ), either the parent or child (but not both) may continue processing the directory stream using *readdir*( ), *rewinddir*( ), or *seekdir*( ). If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of the *d_ino* member is unspecified.

The *readdir*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The *readdir_r*( ) function shall initialize the **dirent** structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dirp*, store a pointer to this structure at the location referenced by *result*, and position the directory stream at the next entry.

The storage pointed to by *entry* shall be large enough for a **dirent** with an array of **char** *d_name* members containing at least {NAME_MAX} plus one elements.

Upon successful return, the pointer returned at **result* shall have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer shall have the value NULL.

The *readdir_r*( ) function shall not return directory entries containing empty names.

If a file is removed from or added to the directory after the most recent call to *opendir*( ) or *rewinddir*( ), whether a subsequent call to *readdir_r*( ) returns an entry for that file is unspecified.

The *readdir_r*( ) function may buffer several directory entries per actual read operation; the *readdir_r*( ) function shall mark for update the *st_atime* field of the directory each time the directory is actually read.

Applications wishing to check for error situations should set *errno* to 0 before calling *readdir*( ). If *errno* is set to non-zero on return, an error occurred.

RETURN VALUE

Upon successful completion, *readdir*( ) shall return a pointer to an object of type **struct dirent**. When an error is encountered, a null pointer shall be returned and *errno* shall be set to indicate the error. When the end of the directory is encountered, a null pointer shall be returned and *errno* is not changed.

TSF If successful, the *readdir_r*( ) function shall return zero; otherwise, an error number shall be returned to indicate the error.

ERRORS

The *readdir*( ) function shall fail if:

[EOVERFLOW] One of the values in the structure to be returned cannot be represented correctly.

The *readdir*( ) function may fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

[ENOENT] The current position of the directory stream is invalid.

The *readdir_r*( ) function may fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

EXAMPLES

The following sample code searches the current directory for the entry *name*:

```
dirp = opendir(".");

while (dirp) {
    errno = 0;
    if ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        }
    } else {
        if (errno == 0) {
            closedir(dirp);
            return NOT_FOUND;
        }
        closedir(dirp);
        return READ_ERROR;
    }
}

return OPEN_ERROR;
```

APPLICATION USAGE

The *readdir*( ) function should be used in conjunction with *opendir*( ), *closedir*( ), and *rewinddir*( ) to examine the contents of the directory.

The *readdir_r*( ) function is thread-safe and shall return values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call.

RATIONALE

The returned value of *readdir*( ) merely *represents* a directory entry. No equivalence should be inferred.

Historical implementations of *readdir*( ) obtain multiple directory entries on a single read operation, which permits subsequent *readdir*( ) operations to operate from the buffered information. Any wording that required each successful *readdir*( ) operation to mark the directory *st_atime* field for update would militate against the historical performance-oriented implementations.

Since *readdir*( ) returns NULL when it detects an error and when the end of the directory is encountered, an application that needs to tell the difference must set *errno* to zero before the call and check it if NULL is returned. Since the function must not change *errno* in the second case and must set it to a non-zero value in the first case, a zero *errno* after a call returning NULL indicates end of directory; otherwise, an error.

Routines to deal with this problem more directly were proposed:

```
int derror (dirp)
DIR *dirp;
```

```
void clearderr (dirp)
DIR *dirp;
```

The first would indicate whether an error had occurred, and the second would clear the error indication. The simpler method involving *errno* was adopted instead by requiring that *readdir*( ) not change *errno* when end-of-directory is encountered.

An error or signal indicating that a directory has changed while open was considered but rejected.

The thread-safe version of the directory reading function returns values in a user-supplied buffer instead of possibly using a static data area that may be overwritten by each call. Either the {NAME_MAX} compile-time constant or the corresponding *pathconf*( ) option can be used to determine the maximum sizes of returned pathnames.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *lstat*( ), *opendir*( ), *rewinddir*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 5

Large File Summit extensions are added.

The *readdir_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *readdir*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *readdir_r*( ) function is marked as part of the Thread-Safe Functions option.

The Open Group Corrigendum U026/7 is applied, correcting the prototype for *readdir_r*( ).

The Open Group Corrigendum U026/8 is applied, clarifying the wording of the successful return for the *readdir_r*( ) function.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- A statement is added to the DESCRIPTION indicating the disposition of certain fields in **struct dirent** when an entry refers to a symbolic link.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENOENT] optional error condition is added.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *readdir_r*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

readlink — read the contents of a symbolic link

SYNOPSIS

```
#include <unistd.h>

ssize_t readlink(const char *restrict path, char *restrict buf,
    size_t bufsize);
```

DESCRIPTION

The *readlink*( ) function shall place the contents of the symbolic link referred to by *path* in the buffer *buf* which has size *bufsize*. If the number of bytes in the symbolic link is less than *bufsize*, the contents of the remainder of *buf* are unspecified. If the *buf* argument is not large enough to contain the link content, the first *bufsize* bytes shall be placed in *buf*.

If the value of *bufsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

RETURN VALUE

Upon successful completion, *readlink*( ) shall return the count of bytes placed in the buffer. Otherwise, it shall return a value of −1, leave the buffer unchanged, and set *errno* to indicate the error.

ERRORS

The *readlink*( ) function shall fail if:

[EACCES]	Search permission is denied for a component of the path prefix of *path*.
[EINVAL]	The *path* argument names a file that is not a symbolic link.
[EIO]	An I/O error occurred while reading from the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

The *readlink*( ) function may fail if:

[EACCES]	Read permission is denied for the directory.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Reading the Name of a Symbolic Link

The following example shows how to read the name of a symbolic link named **/modules/pass1**.

```
#include <unistd.h>

char buf[1024];
int len;
...
if ((len = readlink("/modules/pass1", buf, sizeof(buf)-1)) != -1);
buf[len] = '\0';
```

APPLICATION USAGE

Conforming applications should not assume that the returned contents of the symbolic link are null-terminated.

RATIONALE

Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links, there is no requirement that file times be updated by *readlink*( ). The type associated with *bufsiz* is a **size_t** in order to be consistent with both the ISO C standard and the definition of *read*( ). The behavior specified for *readlink*( ) when *bufsiz* is zero represents historical practice. For this case, the standard developers considered a change whereby *readlink*( ) would return the number of non-null bytes contained in the symbolic link with the buffer *buf* remaining unchanged; however, since the **stat** structure member *st_size* value can be used to determine the size of buffer necessary to contain the contents of the symbolic link as returned by *readlink*( ), this proposal was rejected, and the historical practice retained.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *stat*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The return type is changed to **ssize_t**, to align with the IEEE P1003.1a draft standard.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- This function is made mandatory.
- In this function it is possible for the return value to exceed the range of the type **ssize_t** (since **size_t** has a larger range of positive values than **ssize_t**). A sentence restricting the size of the **size_t** object is added to the description to resolve this conflict.

The following changes are made for alignment with the ISO POSIX-1:1996 standard:

- The FUTURE DIRECTIONS section is changed to None.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The **restrict** keyword is added to the *readlink*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

readv — read a vector

SYNOPSIS

XSI

```
#include <sys/uio.h>

ssize_t readv(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The *readv*( ) function shall be equivalent to *read*( ), except as described below. The *readv*( ) function shall place the input data into the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], . . ., *iov*[*iovcnt*−1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}.

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. The *readv*( ) function shall always fill an area completely before proceeding to the next.

Upon successful completion, *readv*( ) shall mark for update the *st_atime* field of the file.

RETURN VALUE

Refer to *read*( ).

ERRORS

Refer to *read*( ).

In addition, the *readv*( ) function shall fail if:

[EINVAL] The sum of the *iov_len* values in the *iov* array overflowed an **ssize_t**.

The *readv*( ) function may fail if:

[EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

EXAMPLES

Reading Data into an Array

The following example reads data from the file associated with the file descriptor *fd* into the buffers specified by members of the *iov* array.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_read;
int fd;
char buf0[20];
char buf1[30];
char buf2[40];
int iovcnt;
struct iovec iov[3];

iov[0].iov_base = buf0;
iov[0].iov_len = sizeof(buf0);
iov[1].iov_base = buf1;
iov[1].iov_len = sizeof(buf1);
iov[2].iov_base = buf2;
iov[2].iov_len = sizeof(buf2);
```

```
...
iovcnt = sizeof(iov) / sizeof(struct iovec);
bytes_read = readv(fd, iov, iovcnt);
...
```

APPLICATION USAGE

None.

RATIONALE

Refer to *read*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

read( ), *writev*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/uio.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

Split out from the *read*( ) reference page.

NAME

realloc — memory reallocator

SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *realloc*( ) function shall change the size of the memory object pointed to by *ptr* to the size specified by *size*. The contents of the object shall remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If *size* is 0 and *ptr* is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

If *ptr* is a null pointer, *realloc*( ) shall be equivalent to *malloc*( ) for the specified size.

If *ptr* does not match a pointer returned earlier by *calloc*( ), *malloc*( ), or *realloc*( ) or if the space has previously been deallocated by a call to *free*( ) or *realloc*( ), the behavior is undefined.

The order and contiguity of storage allocated by successive calls to *realloc*( ) is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned shall point to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned.

RETURN VALUE

Upon successful completion with a size not equal to 0, *realloc*( ) shall return a pointer to the (possibly moved) allocated space. If *size* is 0, either a null pointer or a unique pointer that can be successfully passed to *free*( ) shall be returned. If there is not enough available memory, *realloc*( ) shall return a null pointer CX and set *errno* to [ENOMEM].

ERRORS

The *realloc*( ) function shall fail if:

CX [ENOMEM] Insufficient memory is available.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *free*( ), *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, if there is not enough available memory, the setting of *errno* to [ENOMEM] is added.
- The [ENOMEM] error condition is added.

NAME

realpath — resolve a pathname

SYNOPSIS

```
#include <stdlib.h>

char *realpath(const char *restrict file_name,
    char *restrict resolved_name);
```

DESCRIPTION

The *realpath*( ) function shall derive, from the pathname pointed to by *file_name*, an absolute pathname that names the same file, whose resolution does not involve `'.'`, `'..'`, or symbolic links. The generated pathname shall be stored as a null-terminated string, up to a maximum of {PATH_MAX} bytes, in the buffer pointed to by *resolved_name*.

RETURN VALUE

Upon successful completion, *realpath*( ) shall return a pointer to the resolved name. Otherwise, *realpath*( ) shall return a null pointer and set *errno* to indicate the error, and the contents of the buffer pointed to by *resolved_name* are undefined.

ERRORS

The *realpath*( ) function shall fail if:

[EACCES]	Read or search permission was denied for a component of *file_name*.
[EINVAL]	Either the *file_name* or *resolved_name* argument is a null pointer.
[EIO]	An error occurred while reading from the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *file_name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *file_name* does not name an existing file or *file_name* points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

The *realpath*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
[ENOMEM]	Insufficient storage space is available.

EXAMPLES

Generating an Absolute Pathname

The following example generates an absolute pathname for the file identified by the *symlinkpath* argument. The generated pathname is stored in the *actualpath* array.

```
#include <stdlib.h>
...
char *symlinkpath = "/tmp/symlink/file";
char actualpath [PATH_MAX+1];
char *ptr;

ptr = realpath(symlinkpath, actualpath);
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getcwd( ), *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The **restrict** keyword is added to the *realpath*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

recv — receive a message from a connected socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

DESCRIPTION

The *recv*( ) function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

The *recv*( ) function takes the following arguments:

socket	Specifies the socket file descriptor.
buffer	Points to a buffer where the message should be stored.
length	Specifies the length in bytes of the buffer pointed to by the *buffer* argument.
flags	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_PEEK	Peeks at an incoming message. The data is treated as unread and the next *recv*( ) or similar function shall still return this data.
MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
MSG_WAITALL	On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

The *recv*( ) function shall return the length of the message written to the buffer pointed to by the *buffer* argument. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recv*( ) shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recv*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

RETURN VALUE

Upon successful completion, *recv*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recv*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *recv*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EINTR]	The *recv*( ) function was interrupted by a signal that was caught, before any data was available.
[EINVAL]	The MSG_OOB flag is set and no out-of-band data is available.
[ENOTCONN]	A receive is attempted on a connection-mode socket that is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The specified flags are not supported for this socket type or protocol.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recv*( ) function may fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The *recv*( ) function is equivalent to *recvfrom*( ) with a zero *address_len* argument, and to *read*( ) if no flags are used.

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

poll( ), *read*( ), *recvmsg*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

recvfrom — receive a message from a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recvfrom(int socket, void *restrict buffer, size_t length,
    int flags, struct sockaddr *restrict address,
    socklen_t *restrict address_len);
```

DESCRIPTION

The *recvfrom*( ) function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The *recvfrom*( ) function takes the following arguments:

socket	Specifies the socket file descriptor.	
buffer	Points to the buffer where the message should be stored.	
length	Specifies the length in bytes of the buffer pointed to by the *buffer* argument.	
flags	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:	
	MSG_PEEK	Peeks at an incoming message. The data is treated as unread and the next *recvfrom*( ) or similar function shall still return this data.
	MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
	MSG_WAITALL	On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.
address	A null pointer, or points to a **sockaddr** structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.	
address_len	Specifies the length of the **sockaddr** structure pointed to by the *address* argument.	

The *recvfrom*( ) function shall return the length of the message written to the buffer pointed to by the *buffer* argument. For message-based sockets, such as SOCK_RAW, SOCK_DGRAM, and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffer, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

Not all protocols provide the source address for messages. If the *address* argument is not a null pointer and the protocol provides the source address of messages, the source address of the received message shall be stored in the **sockaddr** structure pointed to by the *address* argument, and the length of this address shall be stored in the object pointed to by the *address_len* argument.

If the actual length of the address is greater than the length of the supplied **sockaddr** structure, the stored address shall be truncated.

If the *address* argument is not a null pointer and the protocol does not provide the source address of messages, the value stored in the object pointed to by *address* is unspecified.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvfrom*( ) shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recvfrom*( ) shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

RETURN VALUE

Upon successful completion, *recvfrom*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvfrom*( ) shall return 0. Otherwise, the function shall return −1 and set *errno* to indicate the error.

ERRORS

The *recvfrom*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EINTR]	A signal interrupted *recvfrom*( ) before any data was available.
[EINVAL]	The MSG_OOB flag is set and no out-of-band data is available.
[ENOTCONN]	A receive is attempted on a connection-mode socket that is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The specified flags are not supported for this socket type.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvfrom*( ) function may fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

poll( ), *read*( ), *recv*( ), *recvmsg*( ), *select*( ) (on page 1742)1 *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

recvmsg — receive a message from a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

DESCRIPTION

The *recvmsg*( ) function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

The *recvmsg*( ) function takes the following arguments:

socket	Specifies the socket file descriptor.
message	Points to a **msghdr** structure, containing both the buffer to store the source address and the buffers for the incoming message. The length and format of the address depend on the address family of the socket. The *msg_flags* member is ignored on input, but may contain meaningful values on output.
flags	Specifies the type of message reception. Values of this argument are formed by logically OR'ing zero or more of the following values:

MSG_OOB	Requests out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
MSG_PEEK	Peeks at the incoming message.
MSG_WAITALL	On SOCK_STREAM sockets this requests that the function block until the full amount of data can be returned. The function may return the smaller amount of data if the socket is a message-based socket, if a signal is caught, if the connection is terminated, if MSG_PEEK was specified, or if an error is pending for the socket.

The *recvmsg*( ) function shall receive messages from unconnected or connected sockets and shall return the length of the message.

The *recvmsg*( ) function shall return the total length of the message. For message-based sockets, such as SOCK_DGRAM and SOCK_SEQPACKET, the entire message shall be read in a single operation. If a message is too long to fit in the supplied buffers, and MSG_PEEK is not set in the *flags* argument, the excess bytes shall be discarded, and MSG_TRUNC shall be set in the *msg_flags* member of the **msghdr** structure. For stream-based sockets, such as SOCK_STREAM, message boundaries shall be ignored. In this case, data shall be returned to the user as soon as it becomes available, and no data shall be discarded.

If the MSG_WAITALL flag is not set, data shall be returned only up to the end of the first message.

If no messages are available at the socket and O_NONBLOCK is not set on the socket's file descriptor, *recvmsg*( ) shall block until a message arrives. If no messages are available at the socket and O_NONBLOCK is set on the socket's file descriptor, *recvmsg*( ) function shall fail and set *errno* to [EAGAIN] or [EWOULDBLOCK].

In the **msghdr** structure, the *msg_name* and *msg_namelen* members specify the source address if the socket is unconnected. If the socket is connected, the *msg_name* and *msg_namelen* members shall be ignored. The *msg_name* member may be a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* fields are used to specify where the received data shall be stored. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Each storage area indicated by *msg_iov* is filled with received data in turn until all of the received data is stored or all of the areas have been filled.

Upon successful completion, the *msg_flags* member of the message header shall be the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:

MSG_EOR	End of record was received (if supported by the protocol).
MSG_OOB	Out-of-band data was received.
MSG_TRUNC	Normal data was truncated.
MSG_CTRUNC	Control data was truncated.

RETURN VALUE

Upon successful completion, *recvmsg*( ) shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, *recvmsg*( ) shall return 0. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *recvmsg*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The socket's file descriptor is marked O_NONBLOCK and no data is waiting to be received; or MSG_OOB is set and no out-of-band data is available and either the socket's file descriptor is marked O_NONBLOCK or the socket does not support blocking to await out-of-band data.
[EBADF]	The *socket* argument is not a valid open file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EINTR]	This function was interrupted by a signal before any data was available.
[EINVAL]	The sum of the *iov_len* values overflows a **ssize_t**, or the MSG_OOB flag is set and no out-of-band data is available.
[EMSGSIZE]	The *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less than or equal to 0, or is greater than {IOV_MAX}.
[ENOTCONN]	A receive is attempted on a connection-mode socket that is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The specified flags are not supported for this socket type.
[ETIMEDOUT]	The connection timed out during connection establishment, or due to a transmission timeout on active connection.

The *recvmsg*( ) function may fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when data is available to be received.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

poll( ), *recv*( ), *recvfrom*( ), *select*( ), *send*( ), *sendmsg*( ), *sendto*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

regcomp, regerror, regexec, regfree — regular expression matching

SYNOPSIS

```
#include <regex.h>

int regcomp(regex_t *restrict preg, const char *restrict pattern, int cflags);
size_t regerror(int errcode, const regex_t *restrict preg,
    char *restrict errbuf, size_t errbuf_size);
int regexec(const regex_t *restrict preg, const char *restrict string,
    size_t nmatch, regmatch_t pmatch[restrict], int eflags);
void regfree(regex_t *preg);
```

DESCRIPTION

These functions interpret *basic* and *extended* regular expressions as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.

The **regex_t** structure is defined in **<regex.h>** and contains at least the following member:

Member Type	Member Name	Description
`size_t`	`re_nsub`	Number of parenthesized subexpressions.

The **regmatch_t** structure is defined in **<regex.h>** and contains at least the following members:

Member Type	Member Name	Description
regoff_t	*rm_so*	Byte offset from start of *string* to start of substring.
regoff_t	*rm_eo*	Byte offset from start of *string* of the first character after the end of substring.

The *regcomp*( ) function shall compile the regular expression contained in the string pointed to by the *pattern* argument and place the results in the structure pointed to by *preg*. The *cflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in the **<regex.h>** header:

REG_EXTENDED	Use Extended Regular Expressions.
REG_ICASE	Ignore case in match. (See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.)
REG_NOSUB	Report only success / fail in *regexec*( ).
REG_NEWLINE	Change the handling of <newline>s, as described in the text.

The default regular expression type for *pattern* is a Basic Regular Expression. The application can specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

If the REG_NOSUB flag was not set in *cflags*, then *regcomp*( ) shall set *re_nsub* to the number of parenthesized subexpressions (delimited by `"\(\)"` in basic regular expressions or `"( )"` in extended regular expressions) found in *pattern*.

The *regexec*( ) function compares the null-terminated string specified by *string* with the compiled regular expression *preg* initialized by a previous call to *regcomp*( ). If it finds a match, *regexec*( ) shall return 0; otherwise, it shall return non-zero indicating either no match or an error. The *eflags* argument is the bitwise-inclusive OR of zero or more of the following flags, which are defined in the **<regex.h>** header:

REG_NOTBOL	The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (`'^'`), when taken as a special character, shall not match the beginning of *string*.
REG_NOTEOL	The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign (`'$'`), when taken as a special character, shall not match the end of *string*.

If *nmatch* is 0 or REG_NOSUB was set in the *cflags* argument to *regcomp*( ), then *regexec*( ) shall ignore the *pmatch* argument. Otherwise, the application shall ensure that the *pmatch* argument points to an array with at least *nmatch* elements, and *regexec*( ) shall fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions of *pattern*: *pmatch*[*i*].*rm_so* shall be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* shall be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch*−1] shall be filled with −1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then *regexec*( ) shall still do the match, but shall record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match. The following rules shall be used to determine which substrings to report in *pmatch* when matching regular expressions:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*[*i*] shall delimit the last such match.

2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch*[*i*] shall be −1. A subexpression does not participate in the match when:

   `'*'` or `"\{\}"` appears immediately after the subexpression in a basic regular expression, or `'*'`, `'?'`, or `"{}"` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times)

   or:

   `'|'` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or non-match of subexpression *i* reported in *pmatch*[*i*] shall be as described in 1. and 2. above, but within the substring reported in *pmatch*[*j*] rather than the whole string. The offsets in *pmatch*[*i*] are still relative to the start of string.

4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are −1, then the pointers in *pmatch*[*i*] shall also be −1.

5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] shall be the byte offset of the character or null terminator immediately following the zero-length string.

If, when *regexec*( ) is called, the locale is different from when the regular expression was compiled, the result is undefined.

If REG_NEWLINE is not set in *cflags*, then a <newline> in *pattern* or *string* shall be treated as an ordinary character. If REG_NEWLINE is set, then <newline> shall be treated as an ordinary character except as follows:

1. A <newline> in *string* shall not be matched by a period outside a bracket expression or by any form of a non-matching list (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions).
2. A circumflex (`'^'`) in *pattern*, when used to specify expression anchoring (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.8, BRE Expression Anchoring), shall match the zero-length string immediately after a <newline> in *string*, regardless of the setting of REG_NOTBOL.
3. A dollar sign (`'$'`) in *pattern*, when used to specify expression anchoring, shall match the zero-length string immediately before a <newline> in *string*, regardless of the setting of REG_NOTEOL.

The *regfree*( ) function frees any memory allocated by *regcomp*( ) associated with *preg*.

The following constants are defined as error return values:

REG_NOMATCH	*regexec*( ) failed to match.
REG_BADPAT	Invalid regular expression.
REG_ECOLLATE	Invalid collating element referenced.
REG_ECTYPE	Invalid character class type referenced.
REG_EESCAPE	Trailing `'\'` in pattern.
REG_ESUBREG	Number in `"\digit"` invalid or in error.
REG_EBRACK	`"[]"` imbalance.
REG_EPAREN	`"\(\)"` or `"()"` imbalance.
REG_EBRACE	`"\{\}"` imbalance.
REG_BADBR	Content of `"\{\}"` invalid: not a number, number too large, more than two numbers, first larger than second.
REG_ERANGE	Invalid endpoint in range expression.
REG_ESPACE	Out of memory.
REG_BADRPT	`'?'`, `'*'`, or `'+'` not preceded by valid regular expression.

The *regerror*( ) function provides a mapping from error codes returned by *regcomp*( ) and *regexec*( ) to unspecified printable strings. It generates a string corresponding to the value of the *errcode* argument, which the application shall ensure is the last non-zero value returned by *regcomp*( ) or *regexec*( ) with the given value of *preg*. If *errcode* is not such a value, the content of the generated string is unspecified.

If *preg* is a null pointer, but *errcode* is a value returned by a previous call to *regexec*( ) or *regcomp*( ), the *regerror*( ) still generates an error string corresponding to the value of *errcode*, but it might not be as detailed under some implementations.

If the *errbuf_size* argument is not 0, *regerror*( ) shall place the generated string into the buffer of size *errbuf_size* bytes pointed to by *errbuf*. If the string (including the terminating null) cannot fit in the buffer, *regerror*( ) shall truncate the string and null-terminates the result.

If *errbuf_size* is 0, *regerror*( ) shall ignore the *errbuf* argument, and return the size of the buffer needed to hold the generated string.

If the *preg* argument to *regexec*( ) or *regfree*( ) is not a compiled regular expression returned by *regcomp*( ), the result is undefined. A *preg* is no longer treated as a compiled regular expression after it is given to *regfree*( ).

RETURN VALUE

Upon successful completion, the *regcomp*( ) function shall return 0. Otherwise, it shall return an integer value indicating an error as described in **<regex.h>**, and the content of *preg* is undefined. If a code is returned, the interpretation shall be as given in **<regex.h>**.

If *regcomp*( ) detects an invalid RE, it may return REG_BADPAT, or it may return one of the error codes that more precisely describes the error.

Upon successful completion, the *regexec*( ) function shall return 0. Otherwise, it shall return REG_NOMATCH to indicate no match.

Upon successful completion, the *regerror*( ) function shall return the number of bytes needed to hold the entire generated string, including the null termination. If the return value is greater than *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

The *regfree*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

```
#include <regex.h>

/*
 * Match string against the extended regular expression in
 * pattern, treating errors as no match.
 *
 * Return 1 for match, 0 for no match.
 */

int
match(const char *string, char *pattern)
{
    int    status;
    regex_t    re;

    if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
        return(0);      /* Report error. */
    }
    status = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);
    if (status != 0) {
        return(0);      /* Report error. */
    }
    return(1);
}
```

The following demonstrates how the REG_NOTBOL flag could be used with *regexec*( ) to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
(void) regcomp (&re, pattern, 0);
/* This call to regexec() finds the first match on the line. */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) {  /* While matches found. */
    /* Substring found between pm.rm_so and pm.rm_eo. */
    /* This call to regexec() finds the next match. */
    error = regexec (&re, buffer + pm.rm_eo, 1, &pm, REG_NOTBOL);
}
```

APPLICATION USAGE

An application could use:

```
regerror(code,preg,(char *)NULL,(size_t)0)
```

to find out how big a buffer is needed for the generated string, *malloc*( ) a buffer to hold the string, and then call *regerror*( ) again to get the string. Alternatively, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use *malloc*( ) to allocate a larger buffer if it finds that this is too small.

To match a pattern as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation, use the *fnmatch*( ) function.

RATIONALE

The *regmatch*( ) function must fill in all *nmatch* elements of *pmatch*, where *nmatch* and *pmatch* are supplied by the application, even if some elements of *pmatch* do not correspond to subexpressions in *pattern*. The application writer should note that there is probably no reason for using a value of *nmatch* that is larger than *preg*−>*re_nsub*+1.

The REG_NEWLINE flag supports a use of RE matching that is needed in some applications like text editors. In such applications, the user supplies an RE asking the application to find a line that matches the given expression. An anchor in such an RE anchors at the beginning or end of any line. Such an application can pass a sequence of <newline>-separated lines to *regexec*( ) as a single long string and specify REG_NEWLINE to *regcomp*( ) to get the desired behavior. The application must ensure that there are no explicit <newline>s in *pattern* if it wants to ensure that any match occurs entirely within a single line.

The REG_NEWLINE flag affects the behavior of *regexec*( ), but it is in the *cflags* parameter to *regcomp*( ) to allow flexibility of implementation. Some implementations will want to generate the same compiled RE in *regcomp*( ) regardless of the setting of REG_NEWLINE and have *regexec*( ) handle anchors differently based on the setting of the flag. Other implementations will generate different compiled REs based on the REG_NEWLINE.

The REG_ICASE flag supports the operations taken by the *grep* −**i** option and the historical implementations of *ex* and *vi*. Including this flag will make it easier for application code to be written that does the same thing as these utilities.

The substrings reported in *pmatch*[ ] are defined using offsets from the start of the string rather than pointers. Since this is a new interface, there should be no impact on historical implementations or applications, and offsets should be just as easy to use as pointers. The change to offsets was made to facilitate future extensions in which the string to be searched is presented to *regexec*( ) in blocks, allowing a string to be searched that is not all in memory at once.

A new type **regoff_t** is used for the elements of *pmatch*[ ] to ensure that the application can represent either the largest possible array in memory (important for an application conforming to the Shell and Utilities volume of IEEE Std 1003.1-200x) or the largest possible file (important for an application using the extension where a file is searched in chunks).

The standard developers rejected the inclusion of a *regsub*( ) function that would be used to do substitutions for a matched RE. While such a routine would be useful to some applications, its utility would be much more limited than the matching function described here. Both RE parsing and substitution are possible to implement without support other than that required by the ISO C standard, but matching is much more complex than substituting. The only difficult part of substitution, given the information supplied by *regexec*( ), is finding the next character in a string when there can be multi-byte characters. That is a much larger issue, and one that needs a more general solution.

The *errno* variable has not been used for error returns to avoid filling the *errno* name space for this feature.

The interface is defined so that the matched substrings *rm_sp* and *rm_ep* are in a separate **regmatch_t** structure instead of in **regex_t**. This allows a single compiled RE to be used simultaneously in several contexts; in *main*( ) and a signal handler, perhaps, or in multiple threads of lightweight processes. (The *preg* argument to *regexec*( ) is declared with type **const**, so the implementation is not permitted to use the structure to store intermediate results.) It also allows an application to request an arbitrary number of substrings from an RE. The number of subexpressions in the RE is reported in *re_nsub* in *preg*. With this change to *regexec*( ), consideration was given to dropping the REG_NOSUB flag since the user can now specify this with a zero *nmatch* argument to *regexec*( ). However, keeping REG_NOSUB allows an implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp*( ) that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if *nmatch* is not zero and if REG_NOSUB is not specified. Note that the **size_t** type, as defined in the ISO C standard, is unsigned, so the description of *regexec*( ) does not need to address negative values of *nmatch*.

REG_NOTBOL was added to allow an application to do repeated searches for the same pattern in a line. If the pattern contains a circumflex character that should match the beginning of a line, then the pattern should only match when matched against the beginning of the line. Without the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches, but in the general case this would require parsing the expression. The need for REG_NOTEOL is not as clear; it was added for symmetry.

The addition of the *regerror*( ) function addresses the historical need for conforming application programs to have access to error information more than ‘‘Function failed to compile/match your RE for unknown reasons’’. |

This interface provides for two different methods of dealing with error conditions. The specific error codes (REG_EBRACE, for example), defined in **<regex.h>**, allow an application to recover from an error if it is so able. Many applications, especially those that use patterns supplied by a user, will not try to deal with specific error cases, but will just use *regerror*( ) to obtain a human-readable error message to present to the user.

The *regerror*( ) function uses a scheme similar to *confstr*( ) to deal with the problem of allocating memory to hold the generated string. The scheme used by *strerror*( ) in the ISO C standard was considered unacceptable since it creates difficulties for multi-threaded applications.

The *preg* argument is provided to *regerror*( ) to allow an implementation to generate a more descriptive message than would be possible with *errcode* alone. An implementation might, for example, save the character offset of the offending character of the pattern in a field of *preg*, and then include that in the generated message string. The implementation may also ignore *preg*.

A REG_FILENAME flag was considered, but omitted. This flag caused *regexec*( ) to match patterns as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.13, Pattern Matching Notation instead of REs. This service is now provided by the *fnmatch*( )

function.

Notice that there is a difference in philosophy between the ISO POSIX-2:1993 standard and IEEE Std 1003.1-200x in how to handle a bad regular expression. The ISO POSIX-2:1993 standard says that many bad constructs produce undefined results, or that the interpretation is undefined. IEEE Std 1003.1-200x, however, says that the interpretation of such REs is unspecified. The term ‘‘undefined’’ means that the action by the application is an error, of similar severity to passing a bad pointer to a function.

The *regcomp*( ) and *regexec*( ) functions are required to accept any null-terminated string as the *pattern* argument. If the meaning of the string is undefined, the behavior of the function is unspecified. IEEE Std 1003.1-200x does not specify how the functions will interpret the pattern; they might return error codes, or they might do pattern matching in some completely unexpected way, but they should not do something like abort the process.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch( ), *glob*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<regex.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The REG_ENOSYS constant is removed.

The **restrict** keyword is added to the *regcomp*( ), *regerror*( ), and *regexec*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

NAME

remainder, remainderf, remainderl — remainder function

SYNOPSIS

```
#include <math.h>

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall return the floating-point remainder $r=x-ny$ when *y* is non-zero. The value *n* is the integral value nearest the exact value x/y. When $|n-x/y|=\frac{1}{2}$, the value *n* is chosen to be even.

The behavior of *remainder*( ) shall be independent of the rounding mode.

RETURN VALUE

Upon successful completion, these functions shall return the floating-point remainder $r=x-ny$ when *y* is non-zero.

MX If *x* or *y* is NaN, a NaN shall be returned.

If *x* is infinite or *y* is 0 and the other is non-NaN, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error	The *x* argument is ±Inf, or the *y* argument is ±0 and the other argument is non-NaN.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs( ), *div*( ), *feclearexcept*( ), *fetestexcept*( ), *ldiv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The *remainder*( ) function is no longer marked as an extension.

The *remainderf*( ) and *remainderl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

remove — remove a file

SYNOPSIS

```
#include <stdio.h>

int remove(const char *path);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *remove*( ) function shall cause the file named by the pathname pointed to by path to be no longer accessible by that name. A subsequent attempt to open that file using that name shall fail, unless it is created anew.

CX If *path* does not name a directory, *remove*(*path*) shall be equivalent to *unlink*(*path*).

If *path* names a directory, *remove*(*path*) shall be equivalent to *rmdir*(*path*).

RETURN VALUE

CX Refer to *rmdir*( ) or *unlink*( ).

ERRORS

CX Refer to *rmdir*( ) or *unlink*( ).

EXAMPLES

Removing Access to a File

The following example shows how to remove access to a file named **/home/cnd/old_mods**.

```
#include <stdio.h>

int status;
...
status = remove("/home/cnd/old_mods");
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

rmdir( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard and the ISO C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION, RETURN VALUE, and ERRORS sections are updated so that if *path* is not a directory, *remove*( ) is equivalent to *unlink*( ), and if it is a directory, it is equivalent to *rmdir*( ).

NAME

remque — remove an element from a queue

SYNOPSIS

XSI
```
#include <search.h>

void remque(void *element);
```

DESCRIPTION

Refer to *insque*( ).

NAME

remquo, remquof, remquol — remainder functions

SYNOPSIS

```
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *remquo*( ), *remquof*( ), and *remquol*( ) functions shall compute the same remainder as the *remainder*( ), *remainderf*( ), and *remainderl*( ) functions, respectively. In the object pointed to by *quo*, they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of x/y, where n is an implementation-defined integer greater than or equal to 3.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

These functions shall return *x* REM *y*.

MX If *x* or *y* is NaN, a NaN shall be returned.

If *x* is ±Inf or *y* is zero and the other argument is non-NaN, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error	The *x* argument is ±Inf, or the *y* argument is ±0 and the other argument is non-NaN.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions are intended for implementing argument reductions which can exploit a few low-order bits of the quotient. Note that *x* may be so large in magnitude relative to *y* that an exact representation of the quotient is not practical.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *remainder*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

rename — rename a file

SYNOPSIS

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *rename*( ) function shall change the name of a file. The *old* argument points to the pathname of the file to be renamed. The *new* argument points to the new pathname of the file.

CX If either the *old* or *new* argument names a symbolic link, *rename*( ) shall operate on the symbolic link itself, and shall not resolve the last component of the argument. If the *old* argument and the *new* argument resolve to the same existing file, *rename*( ) shall return successfully and perform no other action.

If the *old* argument points to the pathname of a file that is not a directory, the *new* argument shall not point to the pathname of a directory. If the link named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall remain visible to other processes throughout the renaming operation and refer either to the file referred to by *new* or *old* before the operation began. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument shall not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it shall be removed and *old* renamed to *new*. In this case, a link named *new* shall exist throughout the renaming operation and shall refer either to the directory referred to by *new* or *old* before the operation began. If *new* names an existing directory, it shall be required to be an empty directory.

If the *old* argument points to a pathname of a symbolic link, the symbolic link shall be renamed. If the *new* argument points to a pathname of a symbolic link, the symbolic link shall be removed.

The *new* pathname shall not contain a path prefix that names *old*. Write access permission is required for the directory containing *old* and the directory containing *new*. If the *old* argument points to the pathname of a directory, write access permission may be required for the directory named by *old*, and, if it exists, the directory named by *new*.

If the link named by the *new* argument exists and the file's link count becomes 0 when it is removed and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *rename*( ) returns, but the removal of the file contents shall be postponed until all references to the file are closed.

Upon successful completion, *rename*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

If the *rename*( ) function fails for any reason other than [EIO], any file named by *new* shall be unaffected.

RETURN VALUE

CX Upon successful completion, *rename*( ) shall return 0; otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and neither the file named by *old* nor the file named by *new* shall be changed or created.

ERRORS

The *rename*( ) function shall fail if:

CX	[EACCES]	A component of either path prefix denies search permission; or one of the directories containing *old* or *new* denies write permissions; or, write permission is required and is denied for a directory pointed to by the *old* or *new* arguments.
CX	[EBUSY]	The directory named by *old* or *new* is currently in use by the system or another process, and the implementation considers this an error.
CX	[EEXIST] or [ENOTEMPTY]	The link named by *new* is a directory that is not an empty directory.
CX	[EINVAL]	The *new* directory pathname contains a path prefix that names the *old* directory.
CX	[EIO]	A physical I/O error has occurred.
CX	[EISDIR]	The *new* argument points to a directory and the *old* argument points to a file that is not a directory.
CX	[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
CX	[EMLINK]	The file named by *old* is a directory, and the link count of the parent directory of *new* would exceed {LINK_MAX}.
CX	[ENAMETOOLONG]	The length of the *old* or *new* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
CX	[ENOENT]	The link named by *old* does not name an existing file, or either *old* or *new* points to an empty string.
CX	[ENOSPC]	The directory that would contain *new* cannot be extended.
CX	[ENOTDIR]	A component of either path prefix is not a directory; or the *old* argument names a directory and *new* argument names a non-directory file.
XSI	[EPERM] or [EACCES]	The S_ISVTX flag is set on the directory containing the file referred to by *old* and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or *new* refers to an existing file, the S_ISVTX flag is set on the directory containing this file, and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.
CX	[EROFS]	The requested operation requires writing in a directory on a read-only file system.
CX	[EXDEV]	The links named by *new* and *old* are on different file systems and the implementation does not support links between file systems.

The *rename*( ) function may fail if:

XSI	[EBUSY]	The file named by the *old* or *new* arguments is a named STREAM.
CX	[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

CX [ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

CX [ETXTBSY] The file to be renamed is a pure procedure (shared text) file that is being executed.

EXAMPLES

Renaming a File

The following example shows how to rename a file named **/home/cnd/mod1** to **/home/cnd/mod2**.

```
#include <stdio.h>

int status;
...
status = rename("/home/cnd/mod1", "/home/cnd/mod2");
```

APPLICATION USAGE

Some implementations mark for update the *st_ctime* field of renamed files and some do not. Applications which make use of the *st_ctime* field may behave differently with respect to renamed files unless they are designed to allow for either behavior.

RATIONALE

This *rename*( ) function is equivalent for regular files to that defined by the ISO C standard. Its inclusion here expands that definition to include actions on directories and specifies behavior when the *new* parameter names a file that already exists. That specification requires that the action of the function be atomic.

One of the reasons for introducing this function was to have a means of renaming directories while permitting implementations to prohibit the use of *link*( ) and *unlink*( ) with directories, thus constraining links to directories to those made by *mkdir*( ).

The specification that if *old* and *new* refer to the same file is intended to guarantee that:

```
rename("x", "x");
```

does not remove the file.

Renaming dot or dot-dot is prohibited in order to prevent cyclical file system paths.

See also the descriptions of [ENOTEMPTY] and [ENAMETOOLONG] in *rmdir*( ) and [EBUSY] in *unlink*( ). For a discussion of [EXDEV], see *link*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

link( ), *rmdir*( ), *symlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The [EBUSY] error is added to the ‘‘may fail’’ part of the ERRORS section.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

rewind — reset file position indicator in a stream

SYNOPSIS

```
#include <stdio.h>

void rewind(FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The call:

```
rewind(stream)
```

shall be equivalent to:

```
(void) fseek(stream, 0L, SEEK_SET)
```

except that *rewind*( ) shall also clear the error indicator. |

CX Since *rewind*( ) does not return a value, an application wishing to detect errors should clear *errno*, then call *rewind*( ), and if *errno* is non-zero, assume an error has occurred. |

RETURN VALUE

The *rewind*( ) function shall not return a value.

ERRORS

CX Refer to *fseek*( ) with the exception of [EINVAL] which does not apply.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fseek( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

rewinddir — reset position of directory stream to the beginning of a directory

SYNOPSIS

```
#include <dirent.h>

void rewinddir(DIR *dirp);
```

DESCRIPTION

The *rewinddir*( ) function shall reset the position of the directory stream to which *dirp* refers to the beginning of the directory. It shall also cause the directory stream to refer to the current state of the corresponding directory, as a call to *opendir*( ) would have done. If *dirp* does not refer to a directory stream, the effect is undefined.

After a call to the *fork*( ) function, either the parent or child (but not both) may continue XSI processing the directory stream using *readdir*( ), *rewinddir*( ), or *seekdir*( ). If both the parent and child processes use these functions, the result is undefined.

RETURN VALUE

The *rewinddir*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *rewinddir*( ) function should be used in conjunction with *opendir*( ), *readdir*( ), and *closedir*( ) to examine the contents of the directory. This method is recommended for portability.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *opendir*( ), *readdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>** **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

rindex — character string operations (**LEGACY**)

SYNOPSIS

XSI
```
#include <strings.h>

char *rindex(const char *s, int c);
```

DESCRIPTION

The *rindex*( ) function shall be equivalent to *strrchr*( ).

RETURN VALUE

Refer to *strrchr*( ).

ERRORS

Refer to *strrchr*( ).

EXAMPLES

None.

APPLICATION USAGE

strrchr( ) is preferred over this function.

For maximum portability, it is recommended to replace the function call to *rindex*( ) as follows:

```
#define rindex(a,b) strrchr((a),(b))
```

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

strrchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

NAME

rint, rintf, rintl — round-to-nearest integral value

SYNOPSIS

```
#include <math.h>

double rint(double x);
float rintf(float x);
long double rintl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall return the integral value (represented as a **double**) nearest *x* in the direction of the current rounding mode. The current rounding mode is implementation-defined.

If the current rounding mode rounds toward negative infinity, then *rint*( ) shall be equivalent to *floor*( ). If the current rounding mode rounds toward positive infinity, then *rint*( ) shall be equivalent to *ceil*( ).

These functions differ from the *nearbyint*( ), *nearbyintf*( ), and *nearbyintl*( ) functions only in that they may raise the inexact floating-point exception if the result differs in value from the argument.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the integer (represented as a double precision number) nearest *x* in the direction of the current rounding mode.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *rint*( ), *rintf*( ), and *rintl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

XSI

Range Error	The result would cause an overflow.
	If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

abs( ), *ceil*( ), *feclearexcept*( ), *fetestexcept*( ), *nearbyint*( ), *floor*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *rintf*( ) and *rintl*( ) functions are added.
- The *rint*( ) function is no longer marked as an extension.
- The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

rmdir — remove a directory

SYNOPSIS

```
#include <unistd.h>

int rmdir(const char *path);
```

DESCRIPTION

The *rmdir*( ) function shall remove a directory whose name is given by *path*. The directory shall be removed only if it is an empty directory.

If the directory is the root directory or the current working directory of any process, it is unspecified whether the function succeeds, or whether it shall fail and set *errno* to [EBUSY].

If *path* names a symbolic link, then *rmdir*( ) shall fail and set *errno* to [ENOTDIR].

If the *path* argument refers to a path whose final component is either dot or dot-dot, *rmdir*( ) shall fail.

If the directory's link count becomes 0 and no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible. If one or more processes have the directory open when the last link is removed, the dot and dot-dot entries, if present, shall be removed before *rmdir*( ) returns and no new entries may be created in the directory, but the directory shall not be removed until all references to the directory are closed.

If the directory is not an empty directory, *rmdir*( ) shall fail and set *errno* to [EEXIST] or [ENOTEMPTY].

Upon successful completion, the *rmdir*( ) function shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory.

RETURN VALUE

Upon successful completion, the function *rmdir*( ) shall return 0. Otherwise, –1 shall be returned, and *errno* set to indicate the error. If –1 is returned, the named directory shall not be changed.

ERRORS

The *rmdir*( ) function shall fail if:

[EACCES]	Search permission is denied on a component of the path prefix, or write permission is denied on the parent directory of the directory to be removed.
[EBUSY]	The directory to be removed is currently in use by the system or some process and the implementation considers this to be an error.
[EEXIST] or [ENOTEMPTY]	The *path* argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dot-dot.
[EINVAL]	The *path* argument contains a last component that is dot.
[EIO]	A physical I/O error has occurred.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than

NAME_MAX

[ENOENT]	A component of *path* does not name an existing file, or the *path* argument names a nonexistent directory or points to an empty string.
[ENOTDIR]	A component of *path* is not a directory.
XSI [EPERM] or [EACCES]	The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.
[EROFS]	The directory entry to be removed resides on a read-only file system.

The *rmdir*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

Removing a Directory

The following example shows how to remove a directory named **/home/cnd/mod1**.

```
#include <unistd.h>

int status;
...
status = rmdir("/home/cnd/mod1");
```

APPLICATION USAGE

None.

RATIONALE

The *rmdir*( ) and *rename*( ) functions originated in 4.2 BSD, and they used [ENOTEMPTY] for the condition when the directory to be removed does not exist or *new* already exists. When the 1984 /usr/group standard was published, it contained [EEXIST] instead. When these functions were adopted into System V, the 1984 /usr/group standard was used as a reference. Therefore, several existing applications and implementations support/use both forms, and no agreement could be reached on either value. All implementations are required to supply both [EEXIST] and [ENOTEMPTY] in **<errno.h>** with distinct values, so that applications can use both values in C-language **case** statements.

The meaning of deleting *pathname*/**dot** is unclear, because the name of the file (directory) in the parent directory to be removed is not clear, particularly in the presence of multiple links to a directory.

IEEE Std 1003.1-200x was silent with regard to the behavior of *rmdir*( ) when there are multiple hard links to the directory being removed. The requirement to set *errno* to [EEXIST] or [ENOTEMPTY] clarifies the behavior in this case.

If the process' current working directory is being removed, that should be an allowed error.

Virtually all existing implementations detect [ENOTEMPTY] or the case of dot-dot. The text in Section 2.3 (on page 471) about returning any one of the possible errors permits that behavior to continue. The [ELOOP] error may be returned if more than {SYMLOOP_MAX} symbolic links are encountered during resolution of the *path* argument.

FUTURE DIRECTIONS

None.

SEE ALSO

mkdir( ), *remove*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The DESCRIPTION is updated to indicate the results of naming a symbolic link in *path*.
- The [EIO] mandatory error condition is added.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

NAME

round, roundf, roundl — round to nearest integer value in floating-point format

SYNOPSIS

```
#include <math.h>

double round(double x);
float roundf(float x);
long double roundl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the rounded integer value.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *round*( ), *roundf*( ), and *roundl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions may fail if:

XSI Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

scalb — load exponent of a radix-independent floating-point number

SYNOPSIS

OB XSI

```
#include <math.h>

double scalb(double x, double n);
```

DESCRIPTION

The *scalb*( ) function shall compute $x*r^n$, where r is the radix of the machine's floating-point arithmetic. When r is 2, *scalb*( ) shall be equivalent to *ldexp*( ). The value of r is FLT_RADIX which is defined in **<float.h>**.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, the *scalb*( ) function shall return $x*r^n$.

If x or n is NaN, a NaN shall be returned.

If n is zero, x shall be returned.

If x is ±Inf and n is not −Inf, x shall be returned.

If x is ±0 and n is not +Inf, x shall be returned.

If x is ±0 and n is +Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If x is ±Inf and n is −Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If the result would cause an overflow, a range error shall occur and ±HUGE_VAL (according to the sign of x) shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 shall be returned.

ERRORS

The *scalb*( ) function shall fail if:

Domain Error	If x is zero and n is +Inf, or x is Inf and n is −Inf. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
Range Error	The result would overflow. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

The *scalb*( ) function may fail if:

Range Error	The result underflows. If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

Applications should use either *scalbln*( ), *scalblnf*( ), or *scalblnl*( ) in preference to this function.

IEEE Std 1003.1-200x only defines the behavior for the *scalb*( ) function when the *n* argument is an integer, a NaN, or Inf. The behavior of other values for the *n* argument is unspecified.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *ilogb*( ), *ldexp*( ), *logb*( ), *scalbln*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<float.h>**, **<math.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

This function is marked obsolescent.

Although this function is not part of the ISO/IEC 9899:1999 standard, the RETURN VALUE and ERROR sections are updated to align with the error handling in ISO/IEC 9899:1999 standard.

NAME

scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl, — compute exponent using FLT_RADIX

SYNOPSIS

```
#include <math.h>

double scalbln(double x, long n);
float scalblnf(float x, long n);
long double scalblnl(long double x, long n);
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute $x * \text{FLT\_RADIX}^n$ efficiently, not normally by computing FLT_RADIX^n explicitly.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return $x * \text{FLT\_RADIX}^n$.

If the result would cause overflow, a range error shall occur and these functions shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (according to the sign of *x*) as appropriate for the return type of the function.

MX If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

If *n* is 0, *x* shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

ERRORS

These functions shall fail if:

Range Error The result overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

Range Error The result underflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

These functions are named so as to avoid conflicting with the historical definition of the *scalb*( ) function from the Single UNIX Specification. The difference is that the *scalb*( ) function has second argument of **double** instead of **int**. The *scalb*( ) function is not part of ISO C standard. The three functions whose second type is **long** are provided because the factor required to scale from the smallest positive floating-point value to the largest finite one, on many implementations, is too large to represent in the minimum-width **int** format.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *scalb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

scalbn, scalbnf, scalbnl, — compute exponent using FLT_RADIX

SYNOPSIS

```
#include <math.h>

double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
```

DESCRIPTION

Refer to *scalbln*( ).

NAME

scanf — convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *restrict format, ... );
```

DESCRIPTION

Refer to *fscanf*( ).

NAME

sched_get_priority_max, sched_get_priority_min — get priority limits (**REALTIME**)

SYNOPSIS

PS

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

DESCRIPTION

The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the appropriate maximum or minimum, respectively, for the scheduling policy specified by *policy*.

The value of *policy* shall be one of the scheduling policy values defined in **<sched.h>**.

RETURN VALUE

If successful, the *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall return the appropriate maximum or minimum values, respectively. If unsuccessful, they shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sched_get_priority_max*( ) and *sched_get_priority_min*( ) functions shall fail if:

[EINVAL] The value of the *policy* parameter does not represent a defined scheduling policy.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_setparam*( ), *sched_getscheduler*( ), *sched_rr_get_interval*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

These functions are marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The [ESRCH] error condition has been removed since these functions do not take a *pid* argument.

NAME

sched_getparam — get scheduling parameters (**REALTIME**)

SYNOPSIS

PS

```
#include <sched.h>

int sched_getparam(pid_t pid, struct sched_param *param);
```

DESCRIPTION

The *sched_getparam*( ) function shall return the scheduling parameters of a process specified by *pid* in the **sched_param** structure pointed to by *param*.

If a process specified by *pid* exists, and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to *pid* shall be returned.

If *pid* is zero, the scheduling parameters for the calling process shall be returned. The behavior of the *sched_getparam*( ) function is unspecified if the value of *pid* is negative.

RETURN VALUE

Upon successful completion, the *sched_getparam*( ) function shall return zero. If the call to *sched_getparam*( ) is unsuccessful, the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_getparam*( ) function shall fail if:

[EPERM]	The requesting process does not have permission to obtain the scheduling parameters of the specified process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getscheduler( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_getparam*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

NAME

sched_getscheduler — get scheduling policy (**REALTIME**)

SYNOPSIS

PS

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

DESCRIPTION

The *sched_getscheduler*( ) function shall return the scheduling policy of the process specified by *pid*. If the value of *pid* is negative, the behavior of the *sched_getscheduler*( ) function is unspecified.

The values that can be returned by *sched_getscheduler*( ) are defined in the **<sched.h>** header. |

If a process specified by *pid* exists, and if the calling process has permission, the scheduling policy shall be returned for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling policy shall be returned for the calling process.

RETURN VALUE

Upon successful completion, the *sched_getscheduler*( ) function shall return the scheduling policy of the specified process. If unsuccessful, the function shall return −1 and set *errno* to indicate the error.

ERRORS

The *sched_getscheduler*( ) function shall fail if:

[EPERM]	The requesting process does not have permission to determine the scheduling policy of the specified process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_getscheduler*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

NAME

sched_rr_get_interval — get execution time limits (**REALTIME**)

SYNOPSIS

PS
```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *interval);
```

DESCRIPTION

The *sched_rr_get_interval*( ) function shall update the **timespec** structure referenced by the *interval* argument to contain the current execution time limit (that is, time quantum) for the process specified by *pid*. If *pid* is zero, the current execution time limit for the calling process shall be returned.

RETURN VALUE

If successful, the *sched_rr_get_interval*( ) function shall return zero. Otherwise, it shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sched_rr_get_interval*( ) function shall fail if:

[ESRCH] No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_get_priority_max*( ), *sched_getscheduler*( ), *sched_setparam*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_rr_get_interval*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

NAME

sched_setparam — set scheduling parameters (**REALTIME**)

SYNOPSIS

PS

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
```

DESCRIPTION

The *sched_setparam*( ) function shall set the scheduling parameters of the process specified by *pid* to the values specified by the **sched_param** structure pointed to by *param*. The value of the *sched_priority* member in the **sched_param** structure shall be any integer within the inclusive priority range for the current scheduling policy of the process specified by *pid*. Higher numerical values for the priority represent higher priorities. If the value of *pid* is negative, the behavior of the *sched_setparam*( ) function is unspecified.

If a process specified by *pid* exists, and if the calling process has permission, the scheduling parameters shall be set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling parameters shall be set for the calling process.

The conditions under which one process has permission to change the scheduling parameters of another process are implementation-defined.

Implementations may require the requesting process to have the appropriate privilege to set its own scheduling parameters or those of another process.

The target process, whether it is running or not running, shall be moved to the tail of the thread list for its priority.

If the priority of the process specified by the *pid* argument is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the *pid* argument shall preempt a lowest priority running process. Similarly, if the process calling *sched_setparam*( ) sets its own priority lower than that of one or more other non-empty process lists, then the process that is the head of the highest priority list shall also preempt the calling process. Thus, in either case, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.

SS If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget* members of the *param* argument shall represent the time parameters to be used by the sporadic server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param* argument shall have no effect on the scheduling behavior. If the scheduling policy of this process is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the effects of these members are implementation-defined; this case includes the SCHED_OTHER policy.

SS If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the result is implementation-defined; this case includes the SCHED_OTHER policy.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions shall have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling parameters changed to the value specified in *param*. Kernel scheduled entities for use by process contention scope threads that are created after this call completes shall inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads may continue to execute while this function call is in the process of changing the scheduling policy for the underlying kernel-scheduled entities used by the process contention scope threads.

RETURN VALUE

If successful, the *sched_setparam*( ) function shall return zero.

If the call to *sched_setparam*( ) is unsuccessful, the priority shall remain unchanged, and the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sched_setparam*( ) function shall fail if:

[EINVAL]	One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified *pid*.
[EPERM]	The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate privilege to invoke *sched_setparam*( ).
[ESRCH]	No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_getscheduler*( ), *sched_setscheduler*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_setparam*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is added.
- Sections describing two-level scheduling and atomicity of the function are added.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

IEEE PASC Interpretation 1003.1 #100 is applied.

NAME

sched_setscheduler — set scheduling policy and parameters (**REALTIME**)

SYNOPSIS

PS

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy,
    const struct sched_param *param);
```

DESCRIPTION

The *sched_setscheduler*( ) function shall set the scheduling policy and scheduling parameters of the process specified by *pid* to *policy* and the parameters specified in the **sched_param** structure pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param** structure shall be any integer within the inclusive priority range for the scheduling policy specified by *policy*. If the value of *pid* is negative, the behavior of the *sched_setscheduler*( ) function is unspecified.

The possible values for the *policy* parameter are defined in the **<sched.h>** header. |

If a process specified by *pid* exists, and if the calling process has permission, the scheduling policy and scheduling parameters shall be set for the process whose process ID is equal to *pid*.

If *pid* is zero, the scheduling policy and scheduling parameters shall be set for the calling process.

The conditions under which one process has the appropriate privilege to change the scheduling parameters of another process are implementation-defined.

Implementations may require that the requesting process have permission to set its own scheduling parameters or those of another process. Additionally, implementation-defined restrictions may apply as to the appropriate privileges required to set a process' own scheduling policy, or another process' scheduling policy, to a particular value.

The *sched_setscheduler*( ) function shall be considered successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by *policy* and the structure pointed to by *param*, respectively. |

SS If the scheduling policy specified by *policy* is SCHED_SPORADIC, the value specified by the *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget* members of the *param* argument shall represent the time parameters used by the sporadic server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

The value of *sched_ss_max_repl* shall be within the inclusive range [1,{SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the scheduling policy specified by *policy* is either SCHED_FIFO or SCHED_RR, the *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param* argument shall have no effect on the scheduling behavior.

The effect of this function on individual threads is dependent on the scheduling contention scope of the threads:

- For threads with system scheduling contention scope, these functions shall have no effect on their scheduling.
- For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters shall not be affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using these functions.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, then the underlying kernel-scheduled entities for the system contention scope threads shall not be affected by these functions.

The underlying kernel-scheduled entities for the process contention scope threads shall have their scheduling policy and associated scheduling parameters changed to the values specified in *policy* and *param*, respectively. Kernel scheduled entities for use by process contention scope threads that are created after this call completes shall inherit their scheduling policy and associated scheduling parameters from the process.

This function is not atomic with respect to other threads in the process. Threads may continue to execute while this function call is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel-scheduled entities used by the process contention scope threads.

RETURN VALUE

Upon successful completion, the function shall return the former scheduling policy of the specified process. If the *sched_setscheduler*( ) function fails to complete successfully, the policy and scheduling paramenters shall remain unchanged, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sched_setscheduler*( ) function shall fail if:

[EINVAL]	The value of the *policy* parameter is invalid, or one or more of the parameters contained in *param* is outside the valid range for the specified scheduling policy.
[EPERM]	The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.
[ESRCH]	No process can be found corresponding to that specified by *pid*.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sched_getparam( ), *sched_getscheduler*( ), *sched_setparam*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sched_setscheduler*( ) function is marked as part of the Process Scheduling option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Process Scheduling option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect of this function on a thread's scheduling parameters is added.
- Sections describing two-level scheduling and atomicity of the function are added.

The SCHED_SPORADIC scheduling policy is added for alignment with IEEE Std 1003.1d-1999.

NAME

sched_yield — yield processor

SYNOPSIS

PS|THR
```
#include <sched.h>

int sched_yield(void);
```

DESCRIPTION

The *sched_yield*( ) function shall force the running thread to relinquish the processor until it again becomes the head of its thread list. It takes no arguments.

RETURN VALUE

The *sched_yield*( ) function shall return 0 if it completes successfully; otherwise, it shall return a value of −1 and set *errno* to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<sched.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The *sched_yield*( ) function is now marked as part of the Process Scheduling and Threads options.

NAME

seed48 — seed uniformly distributed pseudo-random non-negative long integer generator

SYNOPSIS

XSI
```
#include <stdlib.h>

unsigned short *seed48(unsigned short seed16v[3]);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

seekdir — set position of directory stream

SYNOPSIS

XSI
```
#include <dirent.h>

void seekdir(DIR *dirp, long loc);
```

DESCRIPTION

The *seekdir*( ) function shall set the position of the next *readdir*( ) operation on the directory stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to *telldir*( ). The new position reverts to the one associated with the directory stream when *telldir*( ) was performed.

If the value of *loc* was not obtained from an earlier call to *telldir*( ), or if a call to *rewinddir*( ) occurred between the call to *telldir*( ) and the call to *seekdir*( ), the results of subsequent calls to *readdir*( ) are unspecified.

RETURN VALUE

The *seekdir*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The original standard developers perceived that there were restrictions on the use of the *seekdir*( ) and *telldir*( ) functions related to implementation details, and for that reason these functions need not be supported on all POSIX-conforming systems. They are required on implementations supporting the XSI extension.

One of the perceived problems of implementation is that returning to a given point in a directory is quite difficult to describe formally, in spite of its intuitive appeal, when systems that use B-trees, hashing functions, or other similar mechanisms to order their directories are considered. The definition of *seekdir*( ) and *telldir*( ) does not specify whether, when using these interfaces, a given directory entry will be seen at all, or more than once.

On systems not supporting these functions, their capability can sometimes be accomplished by saving a filename found by *readdir*( ) and later using *rewinddir*( ) and a loop on *readdir*( ) to relocate the position from which the filename was saved.

FUTURE DIRECTIONS

None.

SEE ALSO

opendir( ), *readdir*( ), *telldir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**, **<stdio.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

NAME

select — synchronous I/O multiplexing

SYNOPSIS

```
#include <sys/time.h>

int select(int nfds, fd_set *restrict readfds,
    fd_set *restrict writefds, fd_set *restrict errorfds,
    struct timeval *restrict timeout);
```

DESCRIPTION

Refer to *pselect*( ).

NAME

sem_close — close a named semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

DESCRIPTION

The *sem_close*( ) function shall indicate that the calling process is finished using the named semaphore indicated by *sem*. The effects of calling *sem_close*( ) for an unnamed semaphore (one created by *sem_init*( )) are undefined. The *sem_close*( ) function shall deallocate (that is, make available for reuse by a subsequent *sem_open*( ) by this process) any system resources allocated by the system for use by this process for this semaphore. The effect of subsequent use of the semaphore indicated by *sem* by this process is undefined. If the semaphore has not been removed with a successful call to *sem_unlink*( ), then *sem_close*( ) has no effect on the state of the semaphore. If the *sem_unlink*( ) function has been successfully invoked for *name* after the most recent call to *sem_open*( ) with O_CREAT for this semaphore, then when all processes that have opened the semaphore close it, the semaphore is no longer accessible.

RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sem_close*( ) function shall fail if:

[EINVAL]	The *sem* argument is not a valid semaphore descriptor.

EXAMPLES

None.

APPLICATION USAGE

The *sem_close*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), *sem_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_close*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

NAME

sem_destroy — destroy an unnamed semaphore (**REALTIME**)

SYNOPSIS

SEM
```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

DESCRIPTION

The *sem_destroy*( ) function shall destroy the unnamed semaphore indicated by *sem*. Only a semaphore that was created using *sem_init*( ) may be destroyed using *sem_destroy*( ); the effect of calling *sem_destroy*( ) with a named semaphore is undefined. The effect of subsequent use of the semaphore *sem* is undefined until *sem* is reinitialized by another call to *sem_init*( ).

It is safe to destroy an initialized semaphore upon which no threads are currently blocked. The effect of destroying a semaphore upon which other threads are currently blocked is undefined.

RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sem_destroy*( ) function shall fail if:

[EINVAL]	The *sem* argument is not a valid semaphore.

The *sem_destroy*( ) function may fail if:

[EBUSY]	There are currently processes blocked on the semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_destroy*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_init*( ), *sem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_destroy*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

NAME

sem_getvalue — get the value of a semaphore (**REALTIME**)

SYNOPSIS

SEM
```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

DESCRIPTION

The *sem_getvalue*( ) function shall update the location referenced by the *sval* argument to have the value of the semaphore referenced by *sem* without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.

If *sem* is locked, then the value returned by *sem_getvalue*( ) is either zero or a negative number whose absolute value represents the number of processes waiting for the semaphore at some unspecified time during the call.

RETURN VALUE

Upon successful completion, the *sem_getvalue*( ) function shall return a value of zero. Otherwise, it shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sem_getvalue*( ) function shall fail if:

[EINVAL]	The *sem* argument does not refer to a valid semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_getvalue*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_getvalue*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

The **restrict** keyword is added to the *sem_getvalue*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

NAME

sem_init — initialize an unnamed semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

DESCRIPTION

The *sem_init*( ) function shall initialize the unnamed semaphore referred to by *sem*. The value of the initialized semaphore shall be *value*. Following a successful call to *sem_init*( ), the semaphore may be used in subsequent calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ). This semaphore shall remain usable until the semaphore is destroyed.

If the *pshared* argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore *sem* can use *sem* for performing *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ) operations.

Only *sem* itself may be used for performing synchronization. The result of referring to copies of *sem* in calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ), is undefined.

If the *pshared* argument is zero, then the semaphore is shared between threads of the process; any thread in this process can use *sem* for performing *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_destroy*( ) operations. The use of the semaphore by threads other than those created in the same process is undefined.

Attempting to initialize an already initialized semaphore results in undefined behavior.

RETURN VALUE

Upon successful completion, the *sem_init*( ) function shall initialize the semaphore in *sem*. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *sem_init*( ) function shall fail if:

[EINVAL]	The *value* argument exceeds {SEM_VALUE_MAX}.
[ENOSPC]	A resource required to initialize the semaphore has been exhausted, or the limit on semaphores ({SEM_NSEMS_MAX}) has been reached.
[EPERM]	The process lacks the appropriate privileges to initialize the semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_init*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

Although this volume of IEEE Std 1003.1-200x fails to specify a successful return value, it is likely that a later version may require the implementation to return a value of zero if the call to *sem_init*( ) is successful.

FUTURE DIRECTIONS

None.

SEE ALSO

sem_destroy( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_init*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

sem_open — initialize and open a named semaphore (**REALTIME**)

SYNOPSIS

SEM
```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

DESCRIPTION

The *sem_open*( ) function shall establish a connection between a named semaphore and a process. Following a call to *sem_open*( ) with semaphore name *name*, the process may reference the semaphore associated with *name* using the address returned from the call. This semaphore may be used in subsequent calls to *sem_wait*( ), *sem_trywait*( ), *sem_post*( ), and *sem_close*( ). The semaphore remains usable by this process until the semaphore is closed by a successful call to *sem_close*( ), *_exit*( ), or one of the *exec* functions.

The *oflag* argument controls whether the semaphore is created or merely accessed by the call to *sem_open*( ). The following flag bits may be set in *oflag*:

O_CREAT	This flag is used to create a semaphore if it does not already exist. If O_CREAT is set and the semaphore already exists, then O_CREAT has no effect, except as noted under O_EXCL. Otherwise, *sem_open*( ) creates a named semaphore. The O_CREAT flag requires a third and a fourth argument: *mode*, which is of type **mode_t**, and *value*, which is of type **unsigned**. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to {SEM_VALUE_MAX}. The user ID of the semaphore is set to the effective user ID of the process; the group ID of the semaphore is set to a system default group ID or to the effective group ID of the process. The permission bits of the semaphore are set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are specified, the effect is unspecified. After the semaphore named *name* has been created by *sem_open*( ) with the O_CREAT flag, other processes can connect to the semaphore by calling *sem_open*( ) with the same value of *name*.
O_EXCL	If O_EXCL and O_CREAT are set, *sem_open*( ) fails if the semaphore *name* exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing *sem_open*( ) with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the effect is undefined. If flags other than O_CREAT and O_EXCL are specified in the *oflag* parameter, the effect is unspecified.

The *name* argument points to a string naming a semaphore object. It is unspecified whether the name appears in the file system and is visible to functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *sem_open*( ) with the same value of *name* shall refer to the same semaphore object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

If a process makes multiple successful calls to *sem_open*( ) with the same value for *name*, the same semaphore address shall be returned for each such successful call, provided that there have been no calls to *sem_unlink*( ) for this semaphore.

References to copies of the semaphore produce undefined results.

RETURN VALUE

Upon successful completion, the *sem_open*( ) function shall return the address of the semaphore. Otherwise, it shall return a value of SEM_FAILED and set *errno* to indicate the error. The symbol SEM_FAILED is defined in the **<semaphore.h>** header. No successful return from *sem_open*( ) shall return the value SEM_FAILED.

ERRORS

If any of the following conditions occur, the *sem_open*( ) function shall return SEM_FAILED and set *errno* to the corresponding value:

[EACCES]	The named semaphore exists and the permissions specified by *oflag* are denied, or the named semaphore does not exist and permission to create the named semaphore is denied.
[EEXIST]	O_CREAT and O_EXCL are set and the named semaphore already exists.
[EINTR]	The *sem_open*( ) operation was interrupted by a signal.
[EINVAL]	The *sem_open*( ) operation is not supported for the given name, or O_CREAT was specified in *oflag* and *value* was greater than {SEM_VALUE_MAX}.
[EMFILE]	Too many semaphore descriptors or file descriptors are currently in use by this process.
[ENAMETOOLONG]	The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENFILE]	Too many semaphores are currently open in the system.
[ENOENT]	O_CREAT is not set and the named semaphore does not exist.
[ENOSPC]	There is insufficient space for the creation of the new named semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_open*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

An earlier version of this volume of IEEE Std 1003.1-200x required an error return value of −1 with the type **sem_t *** for the *sem_open*( ) function, which is not guaranteed to be portable across implementations. The revised text provides the symbolic error code SEM_FAILED to eliminate the type conflict.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_close*( ), *sem_post*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_open*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

sem_post — unlock a semaphore (**REALTIME**)

SYNOPSIS

SEM
```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

DESCRIPTION

The *sem_post*( ) function shall unlock the semaphore referenced by *sem* by performing a semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.

If the value of the semaphore resulting from this operation is zero, then one of the threads blocked waiting for the semaphore shall be allowed to return successfully from its call to *sem_wait*( ). PS If the Process Scheduling option is supported, the thread to be unblocked shall be chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest priority waiting thread shall be unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest shall be unblocked. If the Process Scheduling option is not defined, the choice of a thread to unblock is unspecified.

SS If the Process Sporadic Server option is supported, and the scheduling policy is SCHED_SPORADIC, the semantics are as per SCHED_FIFO above.

The *sem_post*( ) function shall be reentrant with respect to signals and may be invoked from a signal-catching function.

RETURN VALUE

If successful, the *sem_post*( ) function shall return zero; otherwise, the function shall return −1 and set *errno* to indicate the error.

ERRORS

The *sem_post*( ) function shall fail if:

[EINVAL]	The *sem* argument does not refer to a valid semaphore.

EXAMPLES

None.

APPLICATION USAGE

The *sem_post*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_timedwait*( ), *sem_trywait*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_post*( ) function is marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

SCHED_SPORADIC is added to the list of scheduling policies for which the thread that is to be unblocked is specified for alignment with IEEE Std 1003.1d-1999.

NAME

sem_timedwait — lock a semaphore (**ADVANCED REALTIME**)

SYNOPSIS

SEM TMO

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
    const struct timespec *restrict abs_timeout);

```

DESCRIPTION

The *sem_timedwait*( ) function shall lock the semaphore referenced by *sem* as in the *sem_wait*( ) function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a *sem_post*( ) function, this wait shall be terminated when the specified timeout expires.

The timeout shall expire when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

TMR If the Timers option is supported, the timeout shall be based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout shall be based on the system clock as returned by the *time*( ) function. The resolution of the timeout shall be the resolution of the clock on which it is based. The **timespec** data type is defined as a structure in the **<time.h>** header. |

Under no circumstance shall the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* need not be checked if the semaphore can be locked immediately.

RETURN VALUE

The *sem_timedwait*( ) function shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sem_timedwait*( ) function shall fail if:

[EINVAL]	The *sem* argument does not refer to a valid semaphore.
[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than zero or greater than or equal to 1 000 million.
[ETIMEDOUT]	The semaphore could not be locked before the specified timeout expired.

The *sem_timedwait*( ) function may fail if:

[EDEADLK]	A deadlock condition was detected.
[EINTR]	A signal interrupted this function.

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *sem_timedwait*( ) function is part of the Semaphores and Timeouts options and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sem_post( ), *sem_trywait*( ), *sem_wait*( ), *semctl*( ), *semget*( ), *semop*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 6. Derived from IEEE Std 1003.1d-1999.

NAME

sem_trywait, sem_wait — lock a semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

DESCRIPTION

The *sem_trywait*( ) function shall lock the semaphore referenced by *sem* only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, shall does not lock the semaphore.

The *sem_wait*( ) function shall lock the semaphore referenced by *sem* by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread shall not return from the call to *sem_wait*( ) until it either locks the semaphore or the call is interrupted by a signal.

Upon successful return, the state of the semaphore shall be locked and shall remain locked until the *sem_post*( ) function is executed and returns successfully.

The *sem_wait*( ) function is interruptible by the delivery of a signal.

RETURN VALUE

The *sem_trywait*( ) and *sem_wait*( ) functions shall return zero if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore shall be unchanged, and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sem_trywait*( ) and *sem_wait*( ) functions shall fail if:

[EAGAIN]	The semaphore was already locked, so it cannot be immediately locked by the *sem_trywait*( ) operation (*sem_trywait*( ) only).
[EINVAL]	The *sem* argument does not refer to a valid semaphore.

The *sem_trywait*( ) and *sem_wait*( ) functions may fail if:

[EDEADLK]	A deadlock condition was detected.
[EINTR]	A signal interrupted this function.

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions may be subject to priority inversion, as discussed in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.285, Priority Inversion.

The *sem_trywait*( ) and *sem_wait*( ) functions are part of the Semaphores option and need not be provided on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_post*( ), *sem_timedwait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_trywait*( ) and *sem_wait*( ) functions are marked as part of the Semaphores option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option.

The *sem_timedwait*( ) function is added to the SEE ALSO section for alignment with IEEE Std 1003.1d-1999.

NAME

sem_unlink — remove a named semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

DESCRIPTION

The *sem_unlink*( ) function shall remove the semaphore named by the string *name*. If the semaphore named by *name* is currently referenced by other processes, then *sem_unlink*( ) shall have no effect on the state of the semaphore. If one or more processes have the semaphore open when *sem_unlink*( ) is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to *sem_close*( ), *_exit*( ), or *exec*. Calls to *sem_open*( ) to recreate or reconnect to the semaphore refer to a new semaphore after *sem_unlink*( ) is called. The *sem_unlink*( ) call shall not block until all references have been destroyed; it shall return immediately. |

RETURN VALUE

Upon successful completion, the *sem_unlink*( ) function shall return a value of 0. Otherwise, the semaphore shall not be changed and the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sem_unlink*( ) function shall fail if:

[EACCES]	Permission is denied to unlink the named semaphore.
[ENAMETOOLONG]	The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |
[ENOENT]	The named semaphore does not exist.

EXAMPLES

None.

APPLICATION USAGE

The *sem_unlink*( ) function is part of the Semaphores option and need not be available on all implementations.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semget*( ), *semop*( ), *sem_close*( ), *sem_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<semaphore.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *sem_unlink*( ) function is marked as part of the Semaphores option. |

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Semaphores option. |

NAME

sem_wait — lock a semaphore (**REALTIME**)

SYNOPSIS

SEM

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

DESCRIPTION

Refer to *sem_trywait*( ).

NAME

semctl — XSI semaphore control operations

SYNOPSIS

XSI

```
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

DESCRIPTION

The *semctl*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *semctl*( ) function provides a variety of semaphore control operations as specified by *cmd*. The fourth argument is optional and depends upon the operation requested. If required, it is of type **union semun**, which the application shall explicitly declare:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short  *array;
} arg;
```

The following semaphore control operations as specified by *cmd* are executed with respect to the semaphore specified by *semid* and *semnum*. The level of permission required for each operation is shown with each command; see Section 2.7 (on page 489). The symbolic names for the values of *cmd* are defined in the **<sys/sem.h>** header:

GETVAL	Return the value of *semval*; see **<sys/sem.h>**. Requires read permission.
SETVAL	Set the value of *semval* to *arg.val*, where *arg* is the value of the fourth argument to *semctl*( ). When this command is successfully executed, the *semadj* value corresponding to the specified semaphore in all processes is cleared. Requires alter permission; see Section 2.7 (on page 489).
GETPID	Return the value of *sempid*. Requires read permission.
GETNCNT	Return the value of *semncnt*. Requires read permission.
GETZCNT	Return the value of *semzcnt*. Requires read permission.

The following values of *cmd* operate on each *semval* in the set of semaphores:

GETALL	Return the value of *semval* for each semaphore in the semaphore set and place into the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl*( ). Requires read permission.
SETALL	Set the value of *semval* for each semaphore in the semaphore set according to the array pointed to by *arg.array*, where *arg* is the fourth argument to *semctl*( ). When this command is successfully executed, the *semadj* values corresponding to each specified semaphore in all processes are cleared. Requires alter permission.

The following values of *cmd* are also available:

IPC_STAT	Place the current value of each member of the **semid_ds** data structure associated with *semid* into the structure pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ). The contents of this structure are defined in

	<sys/sem.h>. Requires read permission.

IPC_SET	Set the value of the following members of the **semid_ds** data structure associated with *semid* to the corresponding value found in the structure pointed to by *arg.buf*, where *arg* is the fourth argument to *semctl*( ):
	`sem_perm.uid`
	`sem_perm.gid`
	`sem_perm.mode`
	The mode bits specified in Section 2.7.1 (on page 490) are copied into the corresponding bits of the *sem_perm.mode* associated with *semid*. The stored values of any other bits are unspecified.
	This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with *semid*.

IPC_RMID	Remove the semaphore identifier specified by *semid* from the system and destroy the set of semaphores and **semid_ds** data structure associated with it. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *sem_perm.cuid* or *sem_perm.uid* in the **semid_ds** data structure associated with *semid*.

RETURN VALUE

If successful, the value returned by *semctl*( ) depends on *cmd* as follows:

GETVAL	The value of *semval*.
GETPID	The value of *sempid*.
GETNCNT	The value of *semncnt*.
GETZCNT	The value of *semzcnt*.
All others	0.

Otherwise, *semctl*( ) shall return −1 and set *errno* to indicate the error.

ERRORS

The *semctl*( ) function shall fail if:

[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EINVAL]	The value of *semid* is not a valid semaphore identifier, or the value of *semnum* is less than 0 or greater than or equal to *sem_nsems*, or the value of *cmd* is not a valid command.
[EPERM]	The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of *sem_perm.cuid* or *sem_perm.uid* in the data structure associated with *semid*.
[ERANGE]	The argument *cmd* is equal to SETVAL or SETALL and the value to which *semval* is to be set is greater than the system-imposed maximum.

EXAMPLES

None.

APPLICATION USAGE

The fourth parameter in the SYNOPSIS section is now specified as `"..."` in order to avoid a clash with the ISO C standard when referring to the union *semun* (as defined in Issue 3) and for backward compatibility.

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semget( ), *semop*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/sem.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to the APPLICATION USAGE section.

NAME

semget — get set of XSI semaphores

SYNOPSIS

XSI

```
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

DESCRIPTION

The *semget*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *semget*( ) function shall return the semaphore identifier associated with *key*.

A semaphore identifier with its associated **semid_ds** data structure and its associated set of *nsems* semaphores (see **<sys/sem.h>**) is created for *key* if one of the following is true:

- The argument *key* is equal to IPC_PRIVATE.
- The argument *key* does not already have a semaphore identifier associated with it and (*semflg* &IPC_CREAT) is non-zero.

Upon creation, the **semid_ds** data structure associated with the new semaphore identifier is initialized as follows:

- In the operation permissions structure *sem_perm.cuid*, *sem_perm.uid*, *sem_perm.cgid*, and *sem_perm.gid* shall be set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of *sem_perm.mode* shall be set equal to the low-order 9 bits of *semflg*.
- The variable *sem_nsems* shall be set equal to the value of *nsems*.
- The variable *sem_otime* shall be set equal to 0 and *sem_ctime* shall be set equal to the current time.
- The data structure associated with each semaphore in the set shall not be initialized. The *semctl*( ) function with the command SETVAL or SETALL can be used to initialize each semaphore.

RETURN VALUE

Upon successful completion, *semget*( ) shall return a non-negative integer, namely a semaphore identifier; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *semget*( ) function shall fail if:

[EACCES]	A semaphore identifier exists for *key*, but operation permission as specified by the low-order 9 bits of *semflg* would not be granted; see Section 2.7 (on page 489).
[EEXIST]	A semaphore identifier exists for the argument *key* but ((*semflg* &IPC_CREAT) &&(*semflg* &IPC_EXCL)) is non-zero.
[EINVAL]	The value of *nsems* is either less than or equal to 0 or greater than the system-imposed limit, or a semaphore identifier exists for the argument *key*, but the number of semaphores in the set associated with it is less than *nsems* and *nsems* is not equal to 0.

[ENOENT]	A semaphore identifier does not exist for the argument *key* and (*semflg* &IPC_CREAT) is equal to 0.
[ENOSPC]	A semaphore identifier is to be created but the system-imposed limit on the maximum number of allowed semaphores system-wide would be exceeded.

EXAMPLES

Creating a Semaphore Identifier

The following example gets a unique semaphore key using the *ftok*( ) function, then gets a semaphore ID associated with that key using the *semget*( ) function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to *semget*( ). In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces *semget*( ) to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to *semop*( ) to initialize it to the values in the *sbuf* array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to *semget*( ) creates a semaphore identifier that can be used later in the program.

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}

/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {

    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
```

```
                sbuf.sem_op = 2;  /* This is the number of runs without queuing. */
                sbuf.sem_flg = 0;
                if (semop(semid, &sbuf, 1) == -1) {
                    perror("IPC error: semop"); exit(1);
                }
            }
            else if (errno == EEXIST) {
                if ((semid = semget(semkey, 0, 0)) == -1) {
                    perror("IPC error 1: semget"); exit(1);
                }
            }
            else {
                perror("IPC error 2: semget"); exit(1);
            }
        }
        ...
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

semctl( ), *semop*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/sem.h>**, Section 2.7 (on page 489).

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

semop — XSI semaphore operations

SYNOPSIS

XSI

```
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

DESCRIPTION

The *semop*( ) function operates on XSI semaphores (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.15, Semaphore). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *semop*( ) function shall perform atomically a user-defined array of semaphore operations on the set of semaphores associated with the semaphore identifier specified by the argument *semid*.

The argument *sops* is a pointer to a user-defined array of semaphore operation structures. The implementation shall not modify elements of this array unless the application uses implementation-defined extensions.

The argument *nsops* is the number of such structures in the array.

Each structure, **sembuf**, includes the following members:

Member Type	Member Name	Description
short	*sem_num*	Semaphore number.
short	*sem_op*	Semaphore operation.
short	*sem_flg*	Operation flags.

Each semaphore operation specified by *sem_op* is performed on the corresponding semaphore specified by *semid* and *sem_num*.

The variable *sem_op* specifies one of three semaphore operations:

1. If *sem_op* is a negative integer and the calling process has alter permission, one of the following shall occur:
   - If *semval*(see **<sys/sem.h>**) is greater than or equal to the absolute value of *sem_op*, the absolute value of *sem_op* is subtracted from *semval*. Also, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be added to the calling process' *semadj* value for the specified semaphore.
   - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is non-zero, *semop*( ) shall return immediately.
   - If *semval* is less than the absolute value of *sem_op* and (*sem_flg* &IPC_NOWAIT) is 0, *semop*( ) shall increment the *semncnt* associated with the specified semaphore and suspend execution of the calling thread until one of the following conditions occurs:
     - — The value of *semval* becomes greater than or equal to the absolute value of *sem_op*. When this occurs, the value of *semncnt* associated with the specified semaphore shall be decremented, the absolute value of *sem_op* shall be subtracted from *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the absolute value of *sem_op* shall be added to the calling process' *semadj* value for the specified semaphore.
     - — The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, *errno* shall be set equal to [EIDRM] and −1 shall be

returned.

- — The calling thread receives a signal that is to be caught. When this occurs, the value of *semncnt* associated with the specified semaphore shall be decremented, and the calling thread shall resume execution in the manner prescribed in *sigaction*( ).

2. If *sem_op* is a positive integer and the calling process has alter permission, the value of *sem_op* shall be added to *semval* and, if (*sem_flg* &SEM_UNDO) is non-zero, the value of *sem_op* shall be subtracted from the calling process' *semadj* value for the specified semaphore.

3. If *sem_op* is 0 and the calling process has read permission, one of the following shall occur:

   - If *semval* is 0, *semop*( ) shall return immediately.
   - If *semval* is non-zero and (*sem_flg* &IPC_NOWAIT) is non-zero, *semop*( ) shall return immediately.
   - If *semval* is non-zero and (*sem_flg* &IPC_NOWAIT) is 0, *semop*( ) shall increment the *semzcnt* associated with the specified semaphore and suspend execution of the calling thread until one of the following occurs:
     - — The value of *semval* becomes 0, at which time the value of *semzcnt* associated with the specified semaphore shall be decremented.
     - — The *semid* for which the calling thread is awaiting action is removed from the system. When this occurs, *errno* shall be set equal to [EIDRM] and −1 shall be returned.
     - — The calling thread receives a signal that is to be caught. When this occurs, the value of *semzcnt* associated with the specified semaphore shall be decremented, and the calling thread shall resume execution in the manner prescribed in *sigaction*( ).

Upon successful completion, the value of *sempid* for each semaphore specified in the array pointed to by *sops* shall be set equal to the process ID of the calling process.

RETURN VALUE

Upon successful completion, *semop*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *semop*( ) function shall fail if:

[E2BIG]	The value of *nsops* is greater than the system-imposed maximum.
[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EAGAIN]	The operation would result in suspension of the calling process but (*sem_flg* &IPC_NOWAIT) is non-zero.
[EFBIG]	The value of *sem_num* is less than 0 or greater than or equal to the number of semaphores in the set associated with *semid*.
[EIDRM]	The semaphore identifier *semid* is removed from the system.
[EINTR]	The *semop*( ) function was interrupted by a signal.
[EINVAL]	The value of *semid* is not a valid semaphore identifier, or the number of individual semaphores for which the calling process requests a SEM_UNDO would exceed the system-imposed limit.

[ENOSPC]	The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.
[ERANGE]	An operation would cause a *semval* to overflow the system-imposed limit, or an operation would cause a *semadj* value to overflow the system-imposed limit.

EXAMPLES

Setting Values in Semaphores

The following example sets the values of the two semaphores associated with the *semid* identifier to the values contained in the *sb* array.

```
#include <sys/sem.h>
...
int semid;
struct sembuf sb[2];
int nsops = 2;
int result;

/* Adjust value of semaphore in the semaphore array semid. */
sb[0].sem_num = 0;
sb[0].sem_op = -1;
sb[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
sb[1].sem_num = 1;
sb[1].sem_op =  1;
sb[1].sem_flg = 0;

result = semop(semid, sb, nsops);
```

Creating a Semaphore Identifier

The following example gets a unique semaphore key using the *ftok*( ) function, then gets a semaphore ID associated with that key using the *semget*( ) function (the first call also tests to make sure the semaphore exists). If the semaphore does not exist, the program creates it, as shown by the second call to *semget*( ). In creating the semaphore for the queuing process, the program attempts to create one semaphore with read/write permission for all. It also uses the IPC_EXCL flag, which forces *semget*( ) to fail if the semaphore already exists.

After creating the semaphore, the program uses a call to *semop*( ) to initialize it to the values in the *sbuf* array. The number of processes that can execute concurrently without queuing is initially set to 2. The final call to *semget*( ) creates a semaphore identifier that can be used later in the program.

The final call to *semop*( ) acquires the semaphore and waits until it is free; the SEM_UNDO option releases the semaphore when the process exits, waiting until there are less than two processes running concurrently.

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <pwd.h>
#include <fcntl.h>
#include <limits.h>
...
key_t semkey;
int semid, pfd, fv;
struct sembuf sbuf;
char *lgn;
char filename[PATH_MAX+1];
struct stat outstat;
struct passwd *pw;
...
/* Get unique key for semaphore. */
if ((semkey = ftok("/tmp", 'a')) == (key_t) -1) {
    perror("IPC error: ftok"); exit(1);
}

/* Get semaphore ID associated with this key. */
if ((semid = semget(semkey, 0, 0)) == -1) {

    /* Semaphore does not exist - Create. */
    if ((semid = semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR |
        S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH)) != -1)
    {
        /* Initialize the semaphore. */
        sbuf.sem_num = 0;
        sbuf.sem_op = 2;  /* This is the number of runs without queuing. */
        sbuf.sem_flg = 0;
        if (semop(semid, &sbuf, 1) == -1) {
            perror("IPC error: semop"); exit(1);
        }
    }
    else if (errno == EEXIST) {
        if ((semid = semget(semkey, 0, 0)) == -1) {
            perror("IPC error 1: semget"); exit(1);
        }
    }
    else {
        perror("IPC error 2: semget"); exit(1);
    }
}
...
sbuf.sem_num = 0;
sbuf.sem_op = -1;
sbuf.sem_flg = SEM_UNDO;
if (semop(semid, &sbuf, 1) == -1) {
    perror("IPC Error: semop"); exit(1);
}
```

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *semctl*( ), *semget*( ), *sem_close*( ), *sem_destroy*( ), *sem_getvalue*( ), *sem_init*( ), *sem_open*( ), *sem_post*( ), *sem_unlink*( ), *sem_wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/ipc.h>**, **<sys/sem.h>**, **<sys/types.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

send — send a message on a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

DESCRIPTION

The *send*( ) function shall initiate transmission of a message from the specified socket to its peer. The *send*( ) function shall send a message only when the socket is connected (including when the peer of a connectionless socket has been set via *connect*( )).

The *send*( ) functions takes the following arguments:

socket	Specifies the socket file descriptor.
buffer	Points to the buffer containing the message to send.
length	Specifies the length of the message in bytes.
flags	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags:

MSG_EOR	Terminates a record (if supported by the protocol).
MSG_OOB	Sends out-of-band data on sockets that support out-of-band communications. The significance and semantics of out-of-band data are protocol-specific.

The length of the message to be sent is specified by the *length* argument. If the message is too long to pass through the underlying protocol, *send*( ) shall fail and no data shall be transmitted.

Successful completion of a call to *send*( ) does not guarantee delivery of the message. A return value of −1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted, and the socket file descriptor does not have O_NONBLOCK set, *send*( ) shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted, and the socket file descriptor does have O_NONBLOCK set, *send*( ) shall fail. The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

The socket in use may require the process to have appropriate privileges to use the *send*( ) function.

RETURN VALUE

Upon successful completion, *send*( ) shall return the number of bytes sent. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *send*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EDESTADDRREQ]	The socket is not connection-mode and no peer address is set.

[EINTR]	A signal interrupted *send*( ) before any data was transmitted.
[EMSGSIZE]	The message is too large be sent all at once, as the socket requires.
[ENOTCONN]	The socket is not connected or otherwise has not had the peer pre-specified.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

The *send*( ) function may fail if:

[EACCES]	The calling process does not have the appropriate privileges.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[ENETDOWN]	The local network interface used to reach the destination is down.
[ENETUNREACH]	No route to the network is present.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.

EXAMPLES

None.

APPLICATION USAGE

The *send*( ) function is equivalent to *sendto*( ) with a null pointer *dest_len* argument, and to *write*( ) if no flags are used. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

connect( ), *getsockopt*( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *sendmsg*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

sendmsg — send a message on a socket using a message structure

SYNOPSIS

```
#include <sys/socket.h>

ssize_t sendmsg(int socket, const struct msghdr *message, int flags);
```

DESCRIPTION

The *sendmsg*( ) function shall send a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message shall be sent to the address specified by **msghdr**. If the socket is connection-mode, the destination address in **msghdr** shall be ignored.

The *sendmsg*( ) function takes the following arguments:

socket	Specifies the socket file descriptor.
message	Points to a **msghdr** structure, containing both the destination address and the buffers for the outgoing message. The length and format of the address depend on the address family of the socket. The *msg_flags* member is ignored.
flags	Specifies the type of message transmission. The application may specify 0 or the following flag:

MSG_EOR	Terminates a record (if supported by the protocol).
MSG_OOB	Sends out-of-band data on sockets that support out-of-bound data. The significance and semantics of out-of-band data are protocol-specific.

The *msg_iov* and *msg_iovlen* fields of *message* specify zero or more buffers containing the data to be sent. *msg_iov* points to an array of **iovec** structures; *msg_iovlen* shall be set to the dimension of this array. In each **iovec** structure, the *iov_base* field specifies a storage area and the *iov_len* field gives its size in bytes. Some of these sizes can be zero. The data from each storage area indicated by *msg_iov* is sent in turn.

Successful completion of a call to *sendmsg*( ) does not guarantee delivery of the message. A return value of −1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, *sendmsg*( ) function shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, *sendmsg*( ) function shall fail.

If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, *sendmsg*( ) shall fail if the SO_BROADCAST option is not set for the socket.

The socket in use may require the process to have appropriate privileges to use the *sendmsg*( ) function.

RETURN VALUE

Upon successful completion, *sendmsg*( ) shall return the number of bytes sent. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sendmsg*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]

The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.

[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EINTR]	A signal interrupted *sendmsg*( ) before any data was transmitted.
[EINVAL]	The sum of the *iov_len* values overflows an **ssize_t**.
[EMSGSIZE]	The message is too large to be sent all at once (as the socket requires), or the *msg_iovlen* member of the **msghdr** structure pointed to by *message* is less than or equal to 0 or is greater than {IOV_MAX}.
[ENOTCONN]	The socket is connection-mode but is not connected.
[ENOTSOCK]	The *socket* argument does not refer a socket.
[EOPNOTSUPP]	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

If the address family of the socket is AF_UNIX, then *sendmsg*( ) shall fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in the socket address.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	A component of the pathname does not name an existing file or the path name is an empty string.
[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.

The *sendmsg*( ) function may fail if:

[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
[EDESTADDRREQ]	The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
[EHOSTUNREACH]	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EISCONN]	A destination address was specified and the socket is already connected.
[ENETDOWN]	The local network interface used to reach the destination is down.

[ENETUNREACH]
No route to the network is present.

[ENOBUFS] Insufficient resources were available in the system to perform the operation.

[ENOMEM] Insufficient memory was available to fulfill the request.

If the address family of the socket is AF_UNIX, then *sendmsg*( ) may fail if:

[ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address. |

[ENAMETOOLONG]
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}. |

EXAMPLES

Done.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

sendto — send a message on a socket

SYNOPSIS

```
#include <sys/socket.h>

ssize_t sendto(int socket, const void *message, size_t length,
    int flags, const struct sockaddr *dest_addr,
    socklen_t dest_len);
```

DESCRIPTION

The *sendto*( ) function shall send a message through a connection-mode or connectionless-mode socket. If the socket is connectionless-mode, the message shall be sent to the address specified by *dest_addr*. If the socket is connection-mode, *dest_addr* shall be ignored.

The *sendto*( ) function takes the following arguments:

socket	Specifies the socket file descriptor.
message	Points to a buffer containing the message to be sent.
length	Specifies the size of the message in bytes.
flags	Specifies the type of message transmission. Values of this argument are formed by logically OR'ing zero or more of the following flags: MSG_EOR Terminates a record (if supported by the protocol). MSG_OOB Sends out-of-band data on sockets that support out-of-band data. The significance and semantics of out-of-band data are protocol-specific.
dest_addr	Points to a **sockaddr** structure containing the destination address. The length and format of the address depend on the address family of the socket.
dest_len	Specifies the length of the **sockaddr** structure pointed to by the *dest_addr* argument.

If the socket protocol supports broadcast and the specified address is a broadcast address for the socket protocol, *sendto*( ) shall fail if the SO_BROADCAST option is not set for the socket.

The *dest_addr* argument specifies the address of the target. The *length* argument specifies the length of the message.

Successful completion of a call to *sendto*( ) does not guarantee delivery of the message. A return value of –1 indicates only locally-detected errors.

If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does not have O_NONBLOCK set, *sendto*( ) shall block until space is available. If space is not available at the sending socket to hold the message to be transmitted and the socket file descriptor does have O_NONBLOCK set, *sendto*( ) shall fail.

The socket in use may require the process to have appropriate privileges to use the *sendto*( ) function.

RETURN VALUE

Upon successful completion, *sendto*( ) shall return the number of bytes sent. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sendto*( ) function shall fail if:

[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EAGAIN] or [EWOULDBLOCK]	The socket's file descriptor is marked O_NONBLOCK and the requested operation would block.
[EBADF]	The *socket* argument is not a valid file descriptor.
[ECONNRESET]	A connection was forcibly closed by a peer.
[EINTR]	A signal interrupted *sendto*( ) before any data was transmitted.
[EMSGSIZE]	The message is too large to be sent all at once, as the socket requires.
[ENOTCONN]	The socket is connection-mode but is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.
[EOPNOTSUPP]	The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.
[EPIPE]	The socket is shut down for writing, or the socket is connection-mode and is no longer connected. In the latter case, and if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling thread.

If the address family of the socket is AF_UNIX, then *sendto*( ) shall fail if:

[EIO]	An I/O error occurred while reading from or writing to the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the pathname in the socket address.
[ENAMETOOLONG]	A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.
[ENOENT]	A component of the pathname does not name an existing file or the pathname is an empty string.
[ENOTDIR]	A component of the path prefix of the pathname in the socket address is not a directory.

The *sendto*( ) function may fail if:

[EACCES]	Search permission is denied for a component of the path prefix; or write access to the named socket is denied.
[EDESTADDRREQ]	The socket is not connection-mode and does not have its peer address set, and no destination address was specified.
[EHOSTUNREACH]	The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).
[EINVAL]	The *dest_len* argument is not a valid length for the address family.
[EIO]	An I/O error occurred while reading from or writing to the file system.

[EISCONN]	A destination address was specified and the socket is already connected. This error may or may not be returned for connection mode sockets.
[ENETDOWN]	The local network interface used to reach the destination is down.
[ENETUNREACH]	No route to the network is present.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

If the address family of the socket is AF_UNIX, then *sendto*( ) may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in the socket address.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

The *select*( ) and *poll*( ) functions can be used to determine when it is possible to send more data.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *poll*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

setbuf — assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf(FILE *restrict stream, char *restrict buf);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

Except that it returns no value, the function call:

```
setbuf(stream, buf)
```

shall be equivalent to:

```
setvbuf(stream, buf, _IOFBF, BUFSIZ)
```

if *buf* is not a null pointer, or to:

```
setvbuf(stream, buf, _IONBF, BUFSIZ)
```

if *buf* is a null pointer.

RETURN VALUE

The *setbuf*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

A common source of error is allocating buffer space as an ‘‘automatic’’ variable in a code block, and then failing to close the stream in the same block.

With *setbuf*( ), allocating a buffer of BUFSIZ bytes does not necessarily imply that all of BUFSIZ bytes are used for the buffer area.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *setvbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The prototype for *setbuf*( ) is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

setcontext — set current user context

SYNOPSIS

XSI

```
#include <ucontext.h>

int setcontext(const ucontext_t *ucp);
```

DESCRIPTION

Refer to *getcontext*( ).

NAME

setegid — set effective group ID

SYNOPSIS

```
#include <unistd.h>

int setegid(gid_t gid);
```

DESCRIPTION

If *gid* is equal to the real group ID or the saved set-group-ID, or if the process has appropriate privileges, *setegid*( ) shall set the effective group ID of the calling process to *gid*; the real group ID, saved set-group-ID, and any supplementary group IDs shall remain unchanged.

The *setegid*( ) function shall not affect the supplementary group list in any way.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setegid*( ) function shall fail if:

[EINVAL]	The value of the *gid* argument is invalid and is not supported by the implementation.
[EPERM]	The process does not have appropriate privileges and *gid* does not match the real group ID or the saved set-group-ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setuid*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

setenv — add or change environment variable

SYNOPSIS

CX
```
#include <stdlib.h>

int setenv(const char *envname, const char *envval, int overwrite);
```

DESCRIPTION

The *setenv*( ) function shall update or add a variable in the environment of the calling process. The *envname* argument points to a string containing the name of an environment variable to be added or altered. The environment variable shall be set to the value to which *envval* points. The function shall fail if *envname* points to a string which contains an '=' character. If the environment variable named by *envname* already exists and the value of *overwrite* is non-zero, the function shall return success and the environment shall be updated. If the environment variable named by *envname* already exists and the value of *overwrite* is zero, the function shall return success and the environment shall remain unchanged.

If the application modifies *environ* or the pointers to which it points, the behavior of *setenv*( ) is undefined. The *setenv*( ) function shall update the list of pointers to which *environ* points.

The strings described by *envname* and *envval* are copied by this function.

The *setenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, zero shall be returned. Otherwise, −1 shall be returned, *errno* set to indicate the error, and the environment shall be unchanged.

ERRORS

The *setenv*( ) function shall fail if:

[EINVAL]	The *name* argument is a null pointer, points to an empty string, or points to a string containing an '=' character.
[ENOMEM]	Insufficient memory was available to add a variable or its value to the environment.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Unanticipated results may occur if *setenv*( ) changes the external variable *environ*. In particular, if the optional *envp* argument to *main*( ) is present, it is not changed, and thus may point to an obsolete copy of the environment (as may any other copy of *environ*). However, other than the aforementioned restriction, the developers of IEEE Std 1003.1-200x intended that the traditional method of walking through the environment by way of the *environ* pointer must be supported.

It was decided that *setenv*( ) should be required by this revision because it addresses a piece of missing functionality, and does not impose a significant burden on the implementor.

There was considerable debate as to whether the System V *putenv*( ) function or the BSD *setenv*( ) function should be required as a mandatory function. The *setenv*( ) function was chosen because it permitted the implementation of *unsetenv*( ) function to delete environmental variables, without specifying an additional interface. The *putenv*( ) function is available as an XSI extension.

The standard developers considered requiring that *setenv*( ) indicate an error when a call to it would result in exceeding {ARG_MAX}. The requirement was rejected since the condition might be temporary, with the application eventually reducing the environment size. The ultimate success or failure depends on the size at the time of a call to *exec*, which returns an indication of this error condition.

FUTURE DIRECTIONS

None.

SEE ALSO

getenv( ), *unsetenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

seteuid — set effective user ID

SYNOPSIS

```
#include <unistd.h>

int seteuid(uid_t uid);
```

DESCRIPTION

If *uid* is equal to the real user ID or the saved set-user-ID, or if the process has appropriate privileges, *seteuid*( ) shall set the effective user ID of the calling process to *uid*; the real user ID and saved set-user-ID shall remain unchanged.

The *seteuid*( ) function shall not affect the supplementary group list in any way.

RETURN VALUE

Upon successful completion, 0 shall be returned; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *seteuid*( ) function shall fail if:

[EINVAL]	The value of the *uid* argument is invalid and is not supported by the implementation.
[EPERM]	The process does not have appropriate privileges and *uid* does not match the real group ID or the saved set-group-ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setuid*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

setgid — set-group-ID

SYNOPSIS

```
#include <unistd.h>

int setgid(gid_t gid);
```

DESCRIPTION

If the process has appropriate privileges, *setgid*( ) shall set the real group ID, effective group ID, and the saved set-group-ID of the calling process to *gid*.

If the process does not have appropriate privileges, but *gid* is equal to the real group ID or the saved set-group-ID, *setgid*( ) shall set the effective group ID to *gid*; the real group ID and saved set-group-ID shall remain unchanged.

The *setgid*( ) function shall not affect the supplementary group list in any way.

Any supplementary group IDs of the calling process shall remain unchanged.

RETURN VALUE

Upon successful completion, 0 is returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setgid*( ) function shall fail if:

[EINVAL]	The value of the *gid* argument is invalid and is not supported by the implementation.
[EPERM]	The process does not have appropriate privileges and *gid* does not match the real group ID or the saved set-group-ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setuid*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setregid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- Functionality associated with _POSIX_SAVED_IDS is now mandated. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effects of *setgid*( ) in processes without appropriate privileges are changed
- A requirement that the supplementary group list is not affected is added.

NAME

setgrent — reset group database to first entry

SYNOPSIS

XSI

```
#include <grp.h>

void setgrent(void);
```

DESCRIPTION

Refer to *endgrent*( ).

NAME

sethostent — network host database functions

SYNOPSIS

```
#include <netdb.h>

void sethostent(int stayopen);
```

DESCRIPTION

Refer to *endhostent*( ).

NAME

setitimer — set value of interval timer

SYNOPSIS

XSI

```
#include <sys/time.h>

int setitimer(int which, const struct itimerval *restrict value,
    struct itimerval *restrict ovalue);
```

DESCRIPTION

Refer to *getitimer*( ).

NAME

setjmp — set jump point for a non-local goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

A call to *setjmp*( ), shall save the calling environment in its *env* argument for later use by *longjmp*( ).

It is unspecified whether *setjmp*( ) is a macro or a function. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name *setjmp*, the behavior is undefined.

An application shall ensure that an invocation of *setjmp*( ) appears in one of the following contexts only:

- The entire controlling expression of a selection or iteration statement
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- The operand of a unary `'!'` operator with the resulting expression being the entire controlling expression of a selection or iteration
- The entire expression of an expression statement (possibly cast to **void**)

If the invocation appears in any other context, the behavior is undefined.

RETURN VALUE

If the return is from a direct invocation, *setjmp*( ) shall return 0. If the return is from a call to *longjmp*( ), *setjmp*( ) shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

In general, *sigsetjmp*( ) is more useful in dealing with errors and interrupts encountered in a low-level subroutine of a program.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

longjmp( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

setkey — set encoding key (**CRYPT**)

SYNOPSIS

XSI
```
#include <stdlib.h>

void setkey(const char *key);
```

DESCRIPTION

The *setkey*( ) function provides access to an implementation-defined encoding algorithm. The argument of *setkey*( ) is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is used by the algorithm. This is the key that shall be used with the algorithm to encode a string *block* passed to *encrypt*( ).

The *setkey*( ) function shall not change the setting of *errno* if successful. An application wishing to check for error situations should set *errno* to 0 before calling *setkey*( ). If *errno* is non-zero on return, an error has occurred.

The *setkey*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

No values are returned.

ERRORS

The *setkey*( ) function shall fail if:

[ENOSYS]	The functionality is not supported on this implementation.

EXAMPLES

None.

APPLICATION USAGE

Decoding need not be implemented in all environments. This is related to government restrictions in some countries on encryption and decryption routines. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of *encrypt*( ) does encoding but not decoding.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

crypt( ), *encrypt*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

NAME

setlocale — set program locale

SYNOPSIS

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *setlocale*( ) function selects the appropriate piece of the program's locale, as specified by the *category* and *locale* arguments, and may be used to change or query the program's entire locale or portions thereof. The value *LC_ALL* for *category* names the program's entire locale; other values for *category* name only a part of the program's locale:

	LC_COLLATE	Affects the behavior of regular expressions and the collation functions.
	LC_CTYPE	Affects the behavior of regular expressions, character classification, character conversion functions, and wide-character functions.
CX	*LC_MESSAGES*	Affects what strings are expected by commands and utilities as affirmative or negative responses.
XSI		It also affects what strings are given by commands and utilities as affirmative or negative responses, and the content of messages.
	LC_MONETARY	Affects the behavior of functions that handle monetary values.
	LC_NUMERIC	Affects the behavior of functions that handle numeric values.
	LC_TIME	Affects the behavior of the time conversion functions.

The *locale* argument is a pointer to a character string containing the required setting of *category*. The contents of this string are implementation-defined. In addition, the following preset values of *locale* are defined for all settings of *category*:

CX	"POSIX"	Specifies the minimal environment for C-language translation called POSIX locale. If *setlocale*( ) is not invoked, the POSIX locale is the default at entry to *main*( ).
	"C"	Equivalent to `"POSIX"`.
CX	" "	Specifies an implementation-defined native environment. This corresponds to the value of the associated environment variables, *LC_** and *LANG*; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.
	A null pointer	Used to direct *setlocale*( ) to query the current internationalized environment and return the name of the *locale*( ).

THR The locale state is common to all threads within a process.

RETURN VALUE

Upon successful completion, *setlocale*( ) shall return the string associated with the specified category for the new locale. Otherwise, *setlocale*( ) shall return a null pointer and the program's locale is not changed.

A null pointer for *locale* causes *setlocale*( ) to return a pointer to the string associated with the *category* for the program's current locale. The program's locale shall not be changed.

The string returned by *setlocale*( ) is such that a subsequent call with that string and its associated *category* shall restore that part of the program's locale. The application shall not modify the string returned which may be overwritten by a subsequent call to *setlocale*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The following code illustrates how a program can initialize the international environment for one language, while selectively modifying the program's locale such that regular expressions and string operations can be applied to text recorded in a different language:

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

Internationalized programs must call *setlocale*( ) to initiate a specific language operation. This can be done by calling *setlocale*( ) as follows:

```
setlocale(LC_ALL, "");
```

Changing the setting of *LC_MESSAGES* has no effect on catalogs that have already been opened by calls to *catopen*( ).

RATIONALE

The ISO C standard defines a collection of functions to support internationalization. One of the most significant aspects of these functions is a facility to set and query the *international environment*. The international environment is a repository of information that affects the behavior of certain functionality, namely:

1. Character handling
2. Collating
3. Date / time formatting
4. Numeric editing
5. Monetary formatting
6. Messaging

The *setlocale*( ) function provides the application developer with the ability to set all or portions, called *categories*, of the international environment. These categories correspond to the areas of functionality, mentioned above. The syntax for *setlocale*( ) is as follows:

```
char *setlocale(int category, const char *locale);
```

where *category* is the name of one of following categories, namely:

LC_COLLATE
LC_CTYPE
LC_MESSAGES
LC_MONETARY
LC_NUMERIC
LC_TIME

In addition, a special value called *LC_ALL* directs *setlocale*( ) to set all categories.

There are two primary uses of *setlocale*( ):

1. Querying the international environment to find out what it is set to
2. Setting the international environment, or *locale*, to a specific value

The behavior of *setlocale*( ) in these two areas is described below. Since it is difficult to describe the behavior in words, examples are used to illustrate the behavior of specific uses.

To query the international environment, *setlocale*( ) is invoked with a specific category and the NULL pointer as the locale. The NULL pointer is a special directive to *setlocale*( ) that tells it to query rather than set the international environment. The following syntax is used to query the name of the international environment:

```
setlocale({LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, \
    LC_NUMERIC, LC_TIME},(char *) NULL);
```

The *setlocale*( ) function shall return the string corresponding to the current international environment. This value may be used by a subsequent call to *setlocale*( ) to reset the international environment to this value. However, it should be noted that the return value from *setlocale*( ) may be a pointer to a static area within the function and is not guaranteed to remain unchanged (that is, it may be modified by a subsequent call to *setlocale*( )). Therefore, if the purpose of calling *setlocale*( ) is to save the value of the current international environment so it can be changed and reset later, the return value should be copied to an array of **char** in the calling program.

There are three ways to set the international environment with *setlocale*( ):

setlocale(*category*, *string*)
This usage sets a specific *category* in the international environment to a specific value corresponding to the value of the *string*. A specific example is provided below:

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

In this example, all categories of the international environment are set to the locale corresponding to the string `"fr_FR.ISO-8859-1"`, or to the French language as spoken in France using the ISO / IEC 8859-1: 1998 standard codeset.

If the string does not correspond to a valid locale, *setlocale*( ) shall return a NULL pointer and the international environment is not changed. Otherwise, *setlocale*( ) shall return the name of the locale just set.

setlocale(*category*, "C")
The ISO C standard states that one locale must exist on all conforming implementations. The name of the locale is C and corresponds to a minimal international environment needed to support the C programming language.

setlocale(*category*, "")
This sets a specific category to an implementation-defined default. This corresponds to the value of the environment variables.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *isalnum*( ), *isalpha*( ), *isblank*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *iswalnum*( ), *iswalpha*( ), *iswblank*( ), *iswcntrl*( ), *iswctype*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), *isxdigit*( ), *localeconv*( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *nl_langinfo*( ), *printf*( ), *scanf*( ), *setlocale*( ), *strcoll*( ), *strerror*( ), *strfmon*( ), *strtod*( ), *strxfrm*( ), *tolower*( ), *toupper*( ), *towlower*( ), *towupper*( ), *wcscoll*( ), *wcstod*( ), *wcstombs*( ), *wcsxfrm*( ), *wctomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<langinfo.h>**, **<locale.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

setlogmask — set log priority mask

SYNOPSIS

XSI

```
#include <syslog.h>

int setlogmask(int maskpri);
```

DESCRIPTION

Refer to *closelog*( ).

NAME

setnetent — network database function

SYNOPSIS

```
#include <netdb.h>

void setnetent(int stayopen);
```

DESCRIPTION

Refer to *endnetent*( ).

NAME

setpgid — set process group ID for job control

SYNOPSIS

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

DESCRIPTION

The *setpgid*( ) function shall either join an existing process group or create a new process group within the session of the calling process. The process group ID of a session leader shall not change. Upon successful completion, the process group ID of the process with a process ID that matches *pid* shall be set to *pgid*. As a special case, if *pid* is 0, the process ID of the calling process shall be used. Also, if *pgid* is 0, the process group ID of the indicated process shall be used. |

RETURN VALUE

Upon successful completion, *setpgid*( ) shall return 0; otherwise, −1 shall be returned and *errno* shall be set to indicate the error.

ERRORS

The *setpgid*( ) function shall fail if:

[EACCES]	The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed one of the *exec* functions.
[EINVAL]	The value of the *pgid* argument is less than 0, or is not a value supported by the implementation.
[EPERM]	The process indicated by the *pid* argument is a session leader.
[EPERM]	The value of the *pid* argument matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.
[EPERM]	The value of the *pgid* argument is valid but does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgid* argument in the same session as the calling process.
[ESRCH]	The value of the *pid* argument does not match the process ID of the calling process or of a child process of the calling process.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *setpgid*( ) function shall group processes together for the purpose of signaling, placement in foreground or background, and other job control actions. |

The *setpgid*( ) function is similar to the *setpgrp*( ) function of 4.2 BSD, except that 4.2 BSD allowed the specified new process group to assume any value. This presents certain security problems and is more flexible than necessary to support job control.

To provide tighter security, *setpgid*( ) only allows the calling process to join a process group already in use inside its session or create a new process group whose process group ID was equal to its process ID.

When a job control shell spawns a new job, the processes in the job must be placed into a new process group via *setpgid*( ). There are two timing constraints involved in this action:

1. The new process must be placed in the new process group before the appropriate program is launched via one of the *exec* functions.
2. The new process must be placed in the new process group before the shell can correctly send signals to the new process group.

To address these constraints, the following actions are performed. The new processes call *setpgid*( ) to alter their own process groups after *fork*( ) but before *exec*. This satisfies the first constraint. Under 4.3 BSD, the second constraint is satisfied by the synchronization property of *vfork*( ); that is, the shell is suspended until the child has completed the *exec*, thus ensuring that the child has completed the *setpgid*( ). A new version of *fork*( ) with this same synchronization property was considered, but it was decided instead to merely allow the parent shell process to adjust the process group of its child processes via *setpgid*( ). Both timing constraints are now satisfied by having both the parent shell and the child attempt to adjust the process group of the child process; it does not matter which succeeds first.

Since it would be confusing to an application to have its process group change after it began executing (that is, after *exec*), and because the child process would already have adjusted its process group before this, the [EACCES] error was added to disallow this. |

One non-obvious use of *setpgid*( ) is to allow a job control shell to return itself to its original process group (the one in effect when the job control shell was executed). A job control shell does this before returning control back to its parent when it is terminating or suspending itself as a way of restoring its job control ‘‘state’’ back to what its parent would expect. (Note that the original process group of the job control shell typically matches the process group of its parent, but this is not necessarily always the case.)

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getpgrp*( ), *setsid*( ), *tcsetpgrp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The *setpgid*( ) function is mandatory since _POSIX_JOB_CONTROL is required to be defined in this issue. This is a FIPS requirement.

NAME

setpgrp — set process group ID

SYNOPSIS

XSI
```
#include <unistd.h>

pid_t setpgrp(void);
```

DESCRIPTION

If the calling process is not already a session leader, *setpgrp*( ) sets the process group ID of the calling process to the process ID of the calling process. If *setpgrp*( ) creates a new session, then the new session has no controlling terminal.

The *setpgrp*( ) function has no effect when the calling process is a session leader.

RETURN VALUE

Upon completion, *setpgrp*( ) shall return the process group ID.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *getpid*( ), *getsid*( ), *kill*( ), *setpgid*( ), *setsid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X / OPEN UNIX extension to BASE.

NAME

setpriority — set the nice value

SYNOPSIS

XSI

```
#include <sys/resource.h>

int setpriority(int which, id_t who, int nice);
```

DESCRIPTION

Refer to *getpriority*( ).

NAME

setprotoent — network protocol database functions

SYNOPSIS

```
#include <netdb.h>

void setprotoent(int stayopen);
```

DESCRIPTION

Refer to *endprotoent*( ).

NAME

setpwent — user database function

SYNOPSIS

XSI

```
#include <pwd.h>

void setpwent(void);
```

DESCRIPTION

Refer to *endpwent*( ).

NAME

setregid — set real and effective group IDs

SYNOPSIS

XSI

```
#include <unistd.h>

int setregid(gid_t rgid, gid_t egid);
```

DESCRIPTION

The *setregid*( ) function shall set the real and effective group IDs of the calling process.

If *rgid* is –1, the real group ID shall not be changed; if *egid* is –1, the effective group ID shall not be changed.

The real and effective group IDs may be set to different values in the same call.

Only a process with appropriate privileges can set the real group ID and the effective group ID to any valid value.

A non-privileged process can set either the real group ID to the saved set-group-ID from one of the *exec* family of functions, or the effective group ID to the saved set-group-ID or the real group ID.

Any supplementary group IDs of the calling process remain unchanged.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, –1 shall be returned and *errno* set to indicate the error, and neither of the group IDs are changed.

ERRORS

The *setregid*( ) function shall fail if:

[EINVAL]	The value of the *rgid* or *egid* argument is invalid or out-of-range.
[EPERM]	The process does not have appropriate privileges and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved set-group-ID, was requested.

EXAMPLES

None.

APPLICATION USAGE

If a set-group-ID process sets its effective group ID to its real group ID, it can still set its effective group ID back to the saved set-group-ID.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setreuid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the saved set-group-ID can be set by any of the *exec* family of functions, not just *execev*( ).

NAME

setreuid — set real and effective user IDs

SYNOPSIS

XSI

```
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
```

DESCRIPTION

The *setreuid*( ) function shall set the real and effective user IDs of the current process to the values specified by the *ruid* and *euid* arguments. If *ruid* or *euid* is −1, the corresponding effective or real user ID of the current process shall be left unchanged. |

A process with appropriate privileges can set either ID to any value. An unprivileged process can only set the effective user ID if the *euid* argument is equal to either the real, effective, or saved user ID of the process.

It is unspecified whether a process without appropriate privileges is permitted to change the real user ID to match the current real, effective, or saved set-user-ID of the process.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setreuid*( ) function shall fail if:

[EINVAL]	The value of the *ruid* or *euid* argument is invalid or out-of-range.
[EPERM]	The current process does not have appropriate privileges, and either an attempt was made to change the effective user ID to a value other than the real user ID or the saved set-user-ID or an attempt was made to change the real user ID to a value not permitted by the implementation.

EXAMPLES

Setting the Effective User ID to the Real User ID

The following example sets the effective user ID of the calling process to the real user ID, so that files created later will be owned by the current user.

```
#include <unistd.h>
#include <sys/types.h>
...
setreuid(getuid(), getuid());
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getegid( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

setrlimit — control maximum resource consumption

SYNOPSIS

XSI

```
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);

```

DESCRIPTION

Refer to *getrlimit*( ).

NAME

setservent — network services database functions

SYNOPSIS

```
#include <netdb.h>

void setservent(int stayopen);
```

DESCRIPTION

Refer to *endservent*( ).

NAME

setsid — create session and set process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t setsid(void);
```

DESCRIPTION

The *setsid*( ) function shall create a new session, if the calling process is not a process group leader. Upon return the calling process shall be the session leader of this new session, shall be the process group leader of a new process group, and shall have no controlling terminal. The process group ID of the calling process shall be set equal to the process ID of the calling process. The calling process shall be the only process in the new process group and the only process in the new session.

RETURN VALUE

Upon successful completion, *setsid*( ) shall return the value of the new process group ID of the calling process. Otherwise, it shall return (**pid_t**)–1 and set *errno* to indicate the error.

ERRORS

The *setsid*( ) function shall fail if:

[EPERM]	The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *setsid*( ) function is similar to the *setpgrp*( ) function of System V. System V, without job control, groups processes into process groups and creates new process groups via *setpgrp*( ); only one process group may be part of a login session.

Job control allows multiple process groups within a login session. In order to limit job control actions so that they can only affect processes in the same login session, this volume of IEEE Std 1003.1-200x adds the concept of a session that is created via *setsid*( ). The *setsid*( ) function also creates the initial process group contained in the session. Additional process groups can be created via the *setpgid*( ) function. A System V process group would correspond to a POSIX System Interfaces session containing a single POSIX process group. Note that this function requires that the calling process not be a process group leader. The usual way to ensure this is true is to create a new process with *fork*( ) and have it call *setsid*( ). The *fork*( ) function guarantees that the process ID of the new process does not match any existing process group ID.

FUTURE DIRECTIONS

None.

SEE ALSO

getsid( ), *setpgid*( ), *setpgrp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

setsockopt — set the socket options

SYNOPSIS

```
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,
    const void *option_value, socklen_t option_len);
```

DESCRIPTION

The *setsockopt*( ) function shall set the option specified by the *option_name* argument, at the protocol level specified by the *level* argument, to the value pointed to by the *option_value* argument for the socket associated with the file descriptor specified by the *socket* argument.

The *level* argument specifies the protocol level at which the option resides. To set options at the socket level, specify the *level* argument as SOL_SOCKET. To set options at other levels, supply the appropriate *level* identifier for the protocol controlling the option. For example, to indicate that an option is interpreted by the TCP (Transport Control Protocol), set *level* to IPPROTO_TCP as defined in the **<netinet/in.h>** header.

The *option_name* argument specifies a single option to set. The *option_name* argument and any specified options are passed uninterpreted to the appropriate protocol module for interpretations. The **<sys/socket.h>** header defines the socket-level options. The options are as follows:

SO_DEBUG	Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. This option takes an **int** value. This is a Boolean option.
SO_BROADCAST	Permits sending of broadcast messages, if this is supported by the protocol. This option takes an **int** value. This is a Boolean option.
SO_REUSEADDR	Specifies that the rules used in validating addresses supplied to *bind*( ) should allow reuse of local addresses, if this is supported by the protocol. This option takes an **int** value. This is a Boolean option.
SO_KEEPALIVE	Keeps connections active by enabling the periodic transmission of messages, if this is supported by the protocol. This option takes an **int** value. If the connected socket fails to respond to these messages, the connection is broken and threads writing to that socket are notified with a SIGPIPE signal. This is a Boolean option.
SO_LINGER	Lingers on a *close*( ) if data is present. This option controls the action taken when unsent messages queue on a socket and *close*( ) is performed. If SO_LINGER is set, the system shall block the process during *close*( ) until it can transmit the data or until the time expires. If SO_LINGER is not specified, and *close*( ) is issued, the system handles the call in a way that allows the process to continue as quickly as possible. This option takes a **linger** structure, as defined in the **<sys/socket.h>** header, to specify the state of the option and linger interval.
SO_OOBINLINE	Leaves received out-of-band data (data marked urgent) inline. This option takes an **int** value. This is a Boolean option.

SO_SNDBUF	Sets send buffer size. This option takes an **int** value.
SO_RCVBUF	Sets receive buffer size. This option takes an **int** value.
SO_DONTROUTE	Requests that outgoing messages bypass the standard routing facilities. The destination shall be on a directly-connected network, and messages are directed to the appropriate network interface according to the destination address. The effect, if any, of this option depends on what protocol is in use. This option takes an **int** value. This is a Boolean option.
SO_RCVLOWAT	Sets the minimum number of bytes to process for socket input operations. The default value for SO_RCVLOWAT is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. (They may return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that returned; for example, out-of-band data.) This option takes an **int** value. Note that not all implementations allow this option to be set.
SO_RCVTIMEO	Sets the timeout value that specifies the maximum amount of time an input function waits until it completes. It accepts a **timeval** structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. If a receive operation has blocked for this much time without receiving additional data, it shall return with a partial count or *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data is received. The default for this option is zero, which indicates that a receive operation shall not time out. This option takes a **timeval** structure. Note that not all implementations allow this option to be set.
SO_SNDLOWAT	Sets the minimum number of bytes to process for socket output operations. Non-blocking output operations shall process no data if flow control does not allow the smaller of the send low water mark value or the entire request to be processed. This option takes an **int** value. Note that not all implementations allow this option to be set.
SO_SNDTIMEO	Sets the timeout value specifying the amount of time that an output function blocks because flow control prevents data from being sent. If a send operation has blocked for this time, it shall return with a partial count or with *errno* set to [EAGAIN] or [EWOULDBLOCK] if no data is sent. The default for this option is zero, which indicates that a send operation shall not time out. This option stores a **timeval** structure. Note that not all implementations allow this option to be set.

For Boolean options, 0 indicates that the option is disabled and 1 indicates that the option is enabled.

Options at other protocol levels vary in format and name.

RETURN VALUE

Upon successful completion, *setsockopt*( ) shall return 0. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setsockopt*( ) function shall fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.

[EDOM]	The send and receive timeout values are too big to fit into the timeout fields in the socket structure.
[EINVAL]	The specified option is invalid at the specified socket level or the socket has been shut down.
[EISCONN]	The socket is already connected, and a specified option cannot be set while the socket is connected.
[ENOPROTOOPT]	The option is not supported by the protocol.
[ENOTSOCK]	The *socket* argument does not refer to a socket.

The *setsockopt*( ) function may fail if:

[ENOMEM]	There was insufficient memory available for the operation to complete.
[ENOBUFS]	Insufficient resources are available in the system to complete the call.

EXAMPLES

None.

APPLICATION USAGE

The *setsockopt*( ) function provides an application program with the means to control socket behavior. An application program can use *setsockopt*( ) to allocate buffer space, control timeouts, or permit socket data broadcasts. The **<sys/socket.h>** header defines the socket-level options available to *setsockopt*( ).

Options may exist at multiple protocol levels. The SO_ options are always present at the uppermost socket level.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.10 (on page 508), *bind*( ), *endprotoent*( ), *getsockopt*( ), *socket*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

setstate — switch pseudo-random number generator state arrays

SYNOPSIS

XSI

```
#include <stdlib.h>

char *setstate(const char *state);
```

DESCRIPTION

Refer to *initstate*( ).

NAME

setuid — set user ID

SYNOPSIS

```
#include <unistd.h>

int setuid(uid_t uid);
```

DESCRIPTION

If the process has appropriate privileges, *setuid*( ) shall set the real user ID, effective user ID, and the saved set-user-ID of the calling process to *uid*.

If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the saved set-user-ID, *setuid*( ) shall set the effective user ID to *uid*; the real user ID and saved set-user-ID shall remain unchanged.

The *setuid*( ) function shall not affect the supplementary group list in any way.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *setuid*( ) function shall fail, return −1, and set *errno* to the corresponding value if one or more of the following are true:

[EINVAL]	The value of the *uid* argument is invalid and not supported by the implementation.
[EPERM]	The process does not have appropriate privileges and *uid* does not match the real user ID or the saved set-user-ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The various behaviors of the *setuid*( ) and *setgid*( ) functions when called by non-privileged processes reflect the behavior of different historical implementations. For portability, it is recommended that new non-privileged applications use the *seteuid*( ) and *setegid*( ) functions instead.

The saved set-user-ID capability allows a program to regain the effective user ID established at the last *exec* call. Similarly, the saved set-group-ID capability allows a program to regain the effective group ID established at the last *exec* call. These capabilities are derived from System V. Without them, a program might have to run as superuser in order to perform the same functions, because superuser can write on the user's files. This is a problem because such a program can write on any user's files, and so must be carefully written to emulate the permissions of the calling process properly. In System V, these capabilities have traditionally been implemented only via the *setuid*( ) and *setgid*( ) functions for non-privileged processes. The fact that the behavior of those functions was different for privileged processes made them difficult to use. The POSIX.1-1990 standard defined the *setuid*( ) function to behave differently for privileged and unprivileged users. When the caller had the appropriate privilege, the function set the calling process' real user ID, effective user ID, and saved set-user ID on implementations that supported it. When the caller did not have the appropriate privilege, the function set only the effective user ID, subject to permission checks. The former use is generally needed for utilities like *login* and *su*, which are not conforming applications and thus outside the scope of IEEE Std 1003.1-200x. These utilities wish to change the user ID irrevocably to a new value, generally that of an unprivileged user. The latter use is needed for conforming applications that are installed with the set-user-ID bit and need to perform operations using the real user ID.

IEEE Std 1003.1-200x augments the latter functionality with a mandatory feature named _POSIX_SAVED_IDS. This feature permits a set-user-ID application to switch its effective user ID back and forth between the values of its *exec*-time real user ID and effective user ID. Unfortunately, the POSIX.1-1990 standard did not permit a conforming application using this feature to work properly when it happened to be executed with the (implementation-defined) appropriate privilege. Furthermore, the application did not even have a means to tell whether it had this privilege. Since the saved set-user-ID feature is quite desirable for applications, as evidenced by the fact that NIST required it in FIPS 151-2, it has been mandated by IEEE Std 1003.1-200x. However, there are implementors who have been reluctant to support it given the limitation described above.

The 4.3BSD system handles the problem by supporting separate functions: *setuid*( ) (which always sets both the real and effective user IDs, like *setuid*( ) in IEEE Std 1003.1-200x for privileged users), and *seteuid*( ) (which always sets just the effective user ID, like *setuid*( ) in IEEE Std 1003.1-200x for non-privileged users). This separation of functionality into distinct functions seems desirable. 4.3BSD does not support the saved set-user-ID feature. It supports similar functionality of switching the effective user ID back and forth via *setreuid*( ), which permits reversing the real and effective user IDs. This model seems less desirable than the saved set-user-ID because the real user ID changes as a side effect. The current 4.4BSD includes saved effective IDs and uses them for *seteuid*( ) and *setegid*( ) as described above. The *setreuid*( ) and *setregid*( ) functions will be deprecated or removed.

The solution here is:

- Require that all implementations support the functionality of the saved set-user-ID, which is set by the *exec* functions and by privileged calls to *setuid*( ).
- Add the *seteuid*( ) and *setegid*( ) functions as portable alternatives to *setuid*( ) and *setgid*( ) for non-privileged and privileged processes.

Historical systems have provided two mechanisms for a set-user-ID process to change its effective user ID to be the same as its real user ID in such a way that it could return to the original effective user ID: the use of the *setuid*( ) function in the presence of a saved set-user-ID, or the use of the BSD *setreuid*( ) function, which was able to swap the real and effective user IDs. The changes included in IEEE Std 1003.1-200x provide a new mechanism using *seteuid*( ) in conjunction with a saved set-user-ID. Thus, all implementations with the new *seteuid*( ) mechanism will have a saved set-user-ID for each process, and most of the behavior controlled by _POSIX_SAVED_IDS has been changed to agree with the case where the option was defined. The *kill*( ) function is an exception. Implementors of the new *seteuid*( ) mechanism will generally be required to maintain compatibility with the older mechanisms previously supported by their systems. However, compatibility with this use of *setreuid*( ) and with the _POSIX_SAVED_IDS behavior of *kill*( ) is unfortunately complicated. If an implementation with a saved set-user-ID allows a process to use *setreuid*( ) to swap its real and effective user IDs, but were to leave the saved set-user-ID unmodified, the process would then have an effective user ID equal to the original real user ID, and both real and saved set-user-ID would be equal to the original effective user ID. In that state, the real user would be unable to kill the process, even though the effective user ID of the process matches that of the real user, if the *kill*( ) behavior of _POSIX_SAVED_IDS was used. This is obviously not acceptable. The alternative choice, which is used in at least one implementation, is to change the saved set-user-ID to the effective user ID during most calls to *setreuid*( ). The standard developers considered that alternative to be less correct than the retention of the old behavior of *kill*( ) in such systems. Current conforming applications shall accommodate either behavior from *kill*( ), and there appears to be no strong reason for *kill*( ) to check the saved set-user-ID rather than the effective user ID.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *getegid*( ), *geteuid*( ), *getgid*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setregid*( ), *setreuid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The functionality associated with _POSIX_SAVED_IDS is now mandatory. This is a FIPS requirement.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effects of *setuid*( ) in processes without appropriate privileges are changed.
- A requirement that the supplementary group list is not affected is added.

NAME

setutxent — reset user accounting database to first entry

SYNOPSIS

XSI

```
#include <utmpx.h>

void setutxent(void);
```

DESCRIPTION

Refer to *endutxent*( ).

NAME

setvbuf — assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

int setvbuf(FILE *restrict stream, char *restrict buf, int type,
    size_t size);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *setvbuf*( ) function may be used after the stream pointed to by *stream* is associated with an open file but before any other operation (other than an unsuccessful call to *setvbuf*( )) is performed on the stream. The argument *type* determines how *stream* shall be buffered, as follows:

- {_IOFBF} shall cause input/output to be fully buffered.
- {_IOLBF} shall cause input/output to be line buffered.
- {_IONBF} shall cause input/output to be unbuffered.

If *buf* is not a null pointer, the array it points to may be used instead of a buffer allocated by *setvbuf*( ) and the argument *size* specifies the size of the array; otherwise, *size* may determine the size of a buffer allocated by the *setvbuf*( ) function. The contents of the array at any time are unspecified. |

For information about streams, see Section 2.5 (on page 484).

RETURN VALUE

Upon successful completion, *setvbuf*( ) shall return 0. Otherwise, it shall return a non-zero value CX if an invalid value is given for *type* or if the request cannot be honored, and may set *errno* to indicate the error.

ERRORS

The *setvbuf*( ) function may fail if:

CX [EBADF] The file descriptor underlying *stream* is not valid.

EXAMPLES

None.

APPLICATION USAGE

A common source of error is allocating buffer space as an ‘‘automatic’’ variable in a code block, and then failing to close the stream in the same block.

With *setvbuf*( ), allocating a buffer of *size* bytes does not necessarily imply that all of *size* bytes are used for the buffer area.

Applications should note that many implementations only provide line buffering on input from terminal devices.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *setbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *setvbuf*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

shm_open — open a shared memory object (**REALTIME**)

SYNOPSIS

SHM

```
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);

```

DESCRIPTION

The *shm_open*( ) function shall establish a connection between a shared memory object and a file descriptor. It shall create an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. The file descriptor is used by other functions to refer to that shared memory object. The *name* argument points to a string naming a shared memory object. It is unspecified whether the name appears in the file system and is visible to other functions that take pathnames as arguments. The *name* argument conforms to the construction rules for a pathname. If *name* begins with the slash character, then processes calling *shm_open*( ) with the same value of *name* refer to the same shared memory object, as long as that name has not been removed. If *name* does not begin with the slash character, the effect is implementation-defined. The interpretation of slash characters other than the leading slash character in *name* is implementation-defined.

If successful, *shm_open*( ) shall return a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes. It is unspecified whether the file offset is set. The FD_CLOEXEC file descriptor flag associated with the new file descriptor is set.

The file status flags and file access modes of the open file description are according to the value of *oflag*. The *oflag* argument is the bitwise-inclusive OR of the following flags defined in the **<fcntl.h>** header. Applications specify exactly one of the first two values (access modes) below in the value of *oflag*:

O_RDONLY	Open for read access only.
O_RDWR	Open for read or write access.

Any combination of the remaining flags may be specified in the value of *oflag*:

O_CREAT	If the shared memory object exists, this flag has no effect, except as noted under O_EXCL below. Otherwise, the shared memory object is created; the user ID of the shared memory object shall be set to the effective user ID of the process; the group ID of the shared memory object is set to a system default group ID or to the effective group ID of the process. The permission bits of the shared memory object shall be set to the value of the *mode* argument except those set in the file mode creation mask of the process. When bits in *mode* other than the file permission bits are set, the effect is unspecified. The *mode* argument does not affect whether the shared memory object is opened for reading, for writing, or for both. The shared memory object has a size of zero.
O_EXCL	If O_EXCL and O_CREAT are set, *shm_open*( ) fails if the shared memory object exists. The check for the existence of the shared memory object and the creation of the object if it does not exist is atomic with respect to other processes executing *shm_open*( ) naming the same shared memory object with O_EXCL and O_CREAT set. If O_EXCL is set and O_CREAT is not set, the result is undefined.

O_TRUNC	If the shared memory object exists, and it is successfully opened O_RDWR, the object shall be truncated to zero length and the mode and owner shall be unchanged by this function call. The result of using O_TRUNC with O_RDONLY is undefined.

When a shared memory object is created, the state of the shared memory object, including all data associated with the shared memory object, persists until the shared memory object is unlinked and all other references are gone. It is unspecified whether the name and shared memory object state remain valid after a system reboot.

RETURN VALUE

Upon successful completion, the *shm_open*( ) function shall return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *shm_open*( ) function shall fail if:

[EACCES]	The shared memory object exists and the permissions specified by *oflag* are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or O_TRUNC is specified and write permission is denied.
[EEXIST]	O_CREAT and O_EXCL are set and the named shared memory object already exists.
[EINTR]	The *shm_open*( ) operation was interrupted by a signal.
[EINVAL]	The *shm_open*( ) operation is not supported for the given name.
[EMFILE]	Too many file descriptors are currently in use by this process.
[ENAMETOOLONG]	The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}. |
[ENFILE]	Too many shared memory objects are currently open in the system.
[ENOENT]	O_CREAT is not set and the named shared memory object does not exist.
[ENOSPC]	There is insufficient space for the creation of the new shared memory object.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

When the Memory Mapped Files option is supported, the normal *open*( ) call is used to obtain a descriptor to a file to be mapped according to existing practice with *mmap*( ). When the Shared Memory Objects option is supported, the *shm_open*( ) function shall obtain a descriptor to the shared memory object to be mapped. |

There is ample precedent for having a file descriptor represent several types of objects. In the POSIX.1-1990 standard, a file descriptor can represent a file, a pipe, a FIFO, a tty, or a directory. Many implementations simply have an operations vector, which is indexed by the file descriptor type and does very different operations. Note that in some cases the file descriptor passed to generic operations on file descriptors are returned by *open*( ) or *creat*( ) and in some cases returned by alternate functions, such as *pipe*( ). The latter technique is used by *shm_open*( ).

Note that such shared memory objects can actually be implemented as mapped files. In both cases, the size can be set after the open using *ftruncate*( ). The *shm_open*( ) function itself does not create a shared object of a specified size because this would duplicate an extant function that set the size of an object referenced by a file descriptor.

On implementations where memory objects are implemented using the existing file system, the *shm_open*( ) function may be implemented using a macro that invokes *open*( ), and the *shm_unlink*( ) function may be implemented using a macro that invokes *unlink*( ).

For implementations without a permanent file system, the definition of the name of the memory objects is allowed not to survive a system reboot. Note that this allows systems with a permanent file system to implement memory objects as data structures internal to the implementation as well.

On implementations that choose to implement memory objects using memory directly, a *shm_open*( ) followed by a *ftruncate*( ) and *close*( ) can be used to preallocate a shared memory area and to set the size of that preallocation. This may be necessary for systems without virtual memory hardware support in order to ensure that the memory is contiguous.

The set of valid open flags to *shm_open*( ) was restricted to O_RDONLY, O_RDWR, O_CREAT, and O_TRUNC because these could be easily implemented on most memory mapping systems. This volume of IEEE Std 1003.1-200x is silent on the results if the implementation cannot supply the requested file access because of implementation-defined reasons, including hardware ones.

The error conditions [EACCES] and [ENOTSUP] are provided to inform the application that the implementation cannot complete a request.

[EACCES] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode because it conflicts with another requested mode. An example might be that an application desires to open a memory object two times, mapping different areas with different access modes. If the implementation cannot map a single area into a process space in two places, which would be required if different access modes were required for the two areas, then the implementation may inform the application at the time of the second open.

[ENOTSUP] indicates for implementation-defined reasons, probably hardware-related, that the implementation cannot comply with a requested mode at all. An example would be that the hardware of the implementation cannot support write-only shared memory areas.

On all implementations, it may be desirable to restrict the location of the memory objects to specific file systems for performance (such as a RAM disk) or implementation-defined reasons (shared memory supported directly only on certain file systems). The *shm_open*( ) function may be used to enforce these restrictions. There are a number of methods available to the application to determine an appropriate name of the file or the location of an appropriate directory. One way is from the environment via *getenv*( ). Another would be from a configuration file.

This volume of IEEE Std 1003.1-200x specifies that memory objects have initial contents of zero when created. This is consistent with current behavior for both files and newly allocated memory. For those implementations that use physical memory, it would be possible that such implementations could simply use available memory and give it to the process uninitialized. This, however, is not consistent with standard behavior for the uninitialized data area, the stack, and of course, files. Finally, it is highly desirable to set the allocated memory to zero for security reasons. Thus, initializing memory objects to zero is required.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *dup*( ), *exec*, *fcntl*( ), *mmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_unlink*( ), *umask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<fcntl.h>**, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *shm_open*( ) function is marked as part of the Shared Memory Objects option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Shared Memory Objects option.

NAME

shm_unlink — remove a shared memory object (**REALTIME**)

SYNOPSIS

SHM
```
#include <sys/mman.h>

int shm_unlink(const char *name);
```

DESCRIPTION

The *shm_unlink*( ) function shall remove the name of the shared memory object named by the string pointed to by *name*.

If one or more references to the shared memory object exist when the object is unlinked, the name shall be removed before *shm_unlink*( ) returns, but the removal of the memory object contents shall be postponed until all open and map references to the shared memory object have been removed.

Even if the object continues to exist after the last *shm_unlink*( ), reuse of the name shall subsequently cause *shm_open*( ) to behave as if no shared memory object of this name exists (that is, *shm_open*( ) will fail if O_CREAT is not set, or will create a new shared memory object if O_CREAT is set).

RETURN VALUE

Upon successful completion, a value of zero shall be returned. Otherwise, a value of −1 shall be returned and *errno* set to indicate the error. If −1 is returned, the named shared memory object shall not be changed by this function call.

ERRORS

The *shm_unlink*( ) function shall fail if:

[EACCES]	Permission is denied to unlink the named shared memory object.
[ENAMETOOLONG]	The length of the *name* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	The named shared memory object does not exist.

EXAMPLES

None.

APPLICATION USAGE

Names of memory objects that were allocated with *open*( ) are deleted with *unlink*( ) in the usual fashion. Names of memory objects that were allocated with *shm_open*( ) are deleted with *shm_unlink*( ). Note that the actual memory object is not destroyed until the last close and unmap on it have occurred if it was already in use.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *mmap*( ), *munmap*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shm_open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/mman.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *shm_unlink*( ) function is marked as part of the Shared Memory Objects option.

In the DESCRIPTION, text is added to clarify that reusing the same name after a *shm_unlink*( ) will not attach to the old shared memory object.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Shared Memory Objects option.

NAME

shmat — XSI shared memory attach operation

SYNOPSIS

XSI
```
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

DESCRIPTION

The *shmat*( ) function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *shmat*( ) function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the address space of the calling process. The segment is attached at the address specified by one of the following criteria:

- If *shmaddr* is a null pointer, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is non-zero, the segment is attached at the address given by (*shmaddr* −((*uintptr_t*)*shmaddr* %SHMLBA)). The character '%' is the C-language remainder operator.
- If *shmaddr* is not a null pointer and (*shmflg* &SHM_RND) is 0, the segment is attached at the address given by *shmaddr*.
- The segment is attached for reading if (*shmflg* &SHM_RDONLY) is non-zero and the calling process has read permission; otherwise, if it is 0 and the calling process has read and write permission, the segment is attached for reading and writing.

RETURN VALUE

Upon successful completion, *shmat*( ) shall increment the value of *shm_nattch* in the data structure associated with the shared memory ID of the attached shared memory segment and return the segment's start address.

Otherwise, the shared memory segment shall not be attached, *shmat*( ) shall return −1, and *errno* shall be set to indicate the error.

ERRORS

The *shmat*( ) function shall fail if:

[EACCES]	Operation permission is denied to the calling process; see Section 2.7 (on page 489).
[EINVAL]	The value of *shmid* is not a valid shared memory identifier, the *shmaddr* is not a null pointer, and the value of (*shmaddr* −((*uintptr_t*)*shmaddr* %SHMLBA)) is an illegal address for attaching shared memory; or the *shmaddr* is not a null pointer, (*shmflg* &SHM_RND) is 0, and the value of *shmaddr* is an illegal address for attaching shared memory.
[EMFILE]	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
[ENOMEM]	The available data space is not large enough to accommodate the shared memory segment.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *shmctl*( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

Issue 6

The Open Group Corrigendum U021/13 is applied.

NAME

shmctl — XSI shared memory control operations

SYNOPSIS

XSI

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

DESCRIPTION

The *shmctl*( ) function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *shmctl*( ) function provides a variety of shared memory control operations as specified by *cmd*. The following values for *cmd* are available:

IPC_STAT Place the current value of each member of the **shmid_ds** data structure associated with *shmid* into the structure pointed to by *buf*. The contents of the structure are defined in **<sys/shm.h>**.

IPC_SET Set the value of the following members of the **shmid_ds** data structure associated with *shmid* to the corresponding value found in the structure pointed to by *buf*:

`shm_perm.uid`
`shm_perm.gid`
`shm_perm.mode` Low-order nine bits.

IPC_SET can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated with *shmid*.

IPC_RMID Remove the shared memory identifier specified by *shmid* from the system and destroy the shared memory segment and **shmid_ds** data structure associated with it. IPC_RMID can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *shm_perm.cuid* or *shm_perm.uid* in the **shmid_ds** data structure associated with *shmid*.

RETURN VALUE

Upon successful completion, *shmctl*( ) shall return 0; otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

The *shmctl*( ) function shall fail if:

[EACCES] The argument *cmd* is equal to IPC_STAT and the calling process does not have read permission; see Section 2.7 (on page 489).

[EINVAL] The value of *shmid* is not a valid shared memory identifier, or the value of *cmd* is not a valid command.

[EPERM] The argument *cmd* is equal to IPC_RMID or IPC_SET and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of *shm_perm.cuid* or *shm_perm.uid* in the data structure associated with *shmid*.

The *shmctl*( ) function may fail if:

[EOVERFLOW] The *cmd* argument is IPC_STAT and the *gid* or *uid* value is too large to be stored in the structure pointed to by the *buf* argument.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

shmat( ), *shmdt*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

shmdt — XSI shared memory detach operation

SYNOPSIS

XSI

```
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

DESCRIPTION

The *shmdt*( ) function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *shmdt*( ) function detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process.

RETURN VALUE

Upon successful completion, *shmdt*( ) shall decrement the value of *shm_nattch* in the data structure associated with the shared memory ID of the attached shared memory segment and return 0.

Otherwise, the shared memory segment shall not be detached, *shmdt*( ) shall return –1, and *errno* shall be set to indicate the error.

ERRORS

The *shmdt*( ) function shall fail if:

[EINVAL] The value of *shmaddr* is not the data segment start address of a shared memory segment.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *shmat*( ), *shmctl*( ), *shmget*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

shmget — get XSI shared memory segment

SYNOPSIS

XSI

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

DESCRIPTION

The *shmget*( ) function operates on XSI shared memory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.340, Shared Memory Object). It is unspecified whether this function interoperates with the realtime interprocess communication facilities defined in Section 2.8 (on page 491).

The *shmget*( ) function shall return the shared memory identifier associated with *key*.

A shared memory identifier, associated data structure, and shared memory segment of at least *size* bytes (see **<sys/shm.h>**) are created for *key* if one of the following is true:

- The argument *key* is equal to IPC_PRIVATE.
- The argument *key* does not already have a shared memory identifier associated with it and (*shmflg* &IPC_CREAT) is non-zero.

Upon creation, the data structure associated with the new shared memory identifier shall be initialized as follows:

- The values of *shm_perm.cuid*, *shm_perm.uid*, *shm_perm.cgid*, and *shm_perm.gid* are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of *shm_perm.mode* are set equal to the low-order nine bits of *shmflg*. The value of *shm_segsz* is set equal to the value of *size*.
- The values of *shm_lpid*, *shm_nattch*, *shm_atime*, and *shm_dtime* are set equal to 0.
- The value of *shm_ctime* is set equal to the current time.

When the shared memory segment is created, it shall be initialized with all zero values.

RETURN VALUE

Upon successful completion, *shmget*( ) shall return a non-negative integer, namely a shared memory identifier; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *shmget*( ) function shall fail if:

[EACCES]	A shared memory identifier exists for *key* but operation permission as specified by the low-order nine bits of *shmflg* would not be granted; see Section 2.7 (on page 489).
[EEXIST]	A shared memory identifier exists for the argument *key* but (*shmflg* &IPC_CREAT) &&(*shmflg* &IPC_EXCL) is non-zero.
[EINVAL]	A shared memory segment is to be created and the value of size is less than the system-imposed minimum or greater than the system-imposed maximum.
[EINVAL]	No shared memory segment is to be created and a shared memory segment exists for *key* but the size of the segment associated with it is less than *size* and *size* is not 0.

[ENOENT] A shared memory identifier does not exist for the argument *key* and (*shmflg* &IPC_CREAT) is 0.

[ENOMEM] A shared memory identifier and associated shared memory segment shall be created, but the amount of available physical memory is not sufficient to fill the request.

[ENOSPC] A shared memory identifier is to be created, but the system-imposed limit on the maximum number of allowed shared memory identifiers system-wide would be exceeded.

EXAMPLES

None.

APPLICATION USAGE

The POSIX Realtime Extension defines alternative interfaces for interprocess communication. Application developers who need to use IPC should design their applications so that modules using the IPC routines described in Section 2.7 (on page 489) can be easily modified to use the alternative interfaces.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

shmat( ), *shmctl*( ), *shmdt*( ), *shm_open*( ), *shm_unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/shm.h>**, Section 2.7 (on page 489)

CHANGE HISTORY

First released in Issue 2. Derived from Issue 2 of the SVID.

Issue 5

Moved from SHARED MEMORY to BASE.

The note about use of POSIX Realtime Extension IPC routines has been moved from FUTURE DIRECTIONS to a new APPLICATION USAGE section.

NAME

shutdown — shut down socket send and receive operations

SYNOPSIS

```
#include <sys/socket.h>

int shutdown(int socket, int how);
```

DESCRIPTION

The *shutdown*( ) function shall cause all or part of a full-duplex connection on the socket associated with the file descriptor *socket* to be shut down.

The *shutdown*( ) function takes the following arguments:

socket	Specifies the file descriptor of the socket.
how	Specifies the type of shutdown. The values are as follows:

SHUT_RD	Disables further receive operations.
SHUT_WR	Disables further send operations.
SHUT_RDWR	Disables further send and receive operations.

The *shutdown*( ) function disables subsequent send and/or receive operations on a socket, depending on the value of the *how* argument.

RETURN VALUE

Upon successful completion, *shutdown*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *shutdown*( ) function shall fail if:

[EBADF]	The *socket* argument is not a valid file descriptor.
[EINVAL]	The *how* argument is invalid.
[ENOTCONN]	The socket is not connected.
[ENOTSOCK]	The *socket* argument does not refer to a socket.

The *shutdown*( ) function may fail if:

[ENOBUFS]	Insufficient resources were available in the system to perform the operation.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getsockopt( ), *read*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *select*( ), *send*( ), *sendto*( ), *setsockopt*( ), *socket*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

sigaction — examine and change signal action

SYNOPSIS

CX

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
    struct sigaction *restrict oact);
```

DESCRIPTION

The *sigaction*( ) function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument *sig* specifies the signal; acceptable values are defined in **<signal.h>**.

The structure **sigaction**, used to describe an action to be taken, is defined in the **<signal.h>** header to include at least the following members:

Member Type	Member Name	Description
void(*) (int)	*sa_handler*	SIG_DFL, SIG_IGN, or pointer to a function.
sigset_t	*sa_mask*	Additional set of signals to be blocked during execution of signal-catching function.
int	*sa_flags*	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	*sa_sigaction*	Signal-catching function.

The storage occupied by *sa_handler* and *sa_sigaction* may overlap, and a conforming application shall not use both simultaneously.

If the argument *act* is not a null pointer, it points to a structure specifying the action to be associated with the specified signal. If the argument *oact* is not a null pointer, the action previously associated with the signal is stored in the location pointed to by the argument *oact*. If the argument *act* is a null pointer, signal handling is unchanged; thus, the call can be used to enquire about the current handling of a given signal. The SIGKILL and SIGSTOP signals shall not be added to the signal mask using this mechanism; this restriction shall be enforced by the system without causing an error to be indicated.

XSI|RTS

If the SA_SIGINFO flag (see below) is cleared in the *sa_flags* field of the **sigaction** structure, the *sa_handler* field identifies the action to be associated with the specified signal. If the SA_SIGINFO flag is set in the *sa_flags* field, and the implementation supports the Realtime Signals Extension option or the X/Open System Interfaces Extension option, the *sa_sigaction* field specifies a signal-catching function. If the SA_SIGINFO bit is cleared and the *sa_handler* field specifies a signal-catching function, or if the SA_SIGINFO bit is set, the *sa_mask* field identifies a set of signals that shall be added to the signal mask of the thread before the signal-catching function is invoked. If the *sa_handler* field specifies a signal-catching function, the *sa_mask* field identifies a set of signals that shall be added to the process' signal mask before the signal-catching function is invoked.

The *sa_flags* field can be used to modify the behavior of the specified signal.

The following flags, defined in the **<signal.h>** header, can be set in *sa_flags*:

XSI

SA_NOCLDSTOP Do not generate SIGCHLD when children stop or stopped children continue.

If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is not set in *sa_flags*, and the implementation supports the SIGCHLD signal, then a SIGCHLD signal shall be generated for the calling process whenever any of its child processes stop and a SIGCHLD signal may be generated for the calling process whnever any of its stopped child processes are continued. If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in *sa_flags*, then the implementation shall not generate a SIGCHLD signal in this way.

SA_ONSTACK If set and an alternate signal stack has been declared with *sigaltstack*( ) or *sigstack*( ), the signal shall be delivered to the calling process on that stack. Otherwise, the signal shall be delivered on the current stack.

SA_RESETHAND If set, the disposition of the signal shall be reset to SIG_DFL and the SA_SIGINFO flag shall be cleared on entry to the signal handler.

Note: SIGILL and SIGTRAP cannot be automatically reset when delivered; the system silently enforces this restriction.

Otherwise, the disposition of the signal shall not be modified on entry to the signal handler.

In addition, if this flag is set, *sigaction*( ) behaves as if the SA_NODEFER flag were also set.

SA_RESTART This flag affects the behavior of interruptible functions; that is, those specified to fail with *errno* set to [EINTR]. If set, and a function specified as interruptible is interrupted by this signal, the function shall restart and shall not fail with [EINTR] unless otherwise specified. If the flag is not set, interruptible functions interrupted by this signal shall fail with *errno* set to [EINTR].

SA_SIGINFO If cleared and the signal is caught, the signal-catching function shall be entered as:

```
void func(int signo);
```

where *signo* is the only argument to the signal catching function. In this case, the application shall use the *sa_handler* member to describe the signal catching function and the application shall not modify the *sa_sigaction* member.

If SA_SIGINFO is set and the signal is caught, the signal-catching function shall be entered as:

```
void func(int signo, siginfo_t *info, void *context);
```

where two additional arguments are passed to the signal catching function. The second argument shall point to an object of type **siginfo_t** explaining the reason why the signal was generated; the third argument can be cast to a pointer to an object of type **ucontext_t** to refer to the receiving process' context that was interrupted when the signal was delivered. In this case, the application shall use the *sa_sigaction* member to describe the signal catching function and the application shall not modify the *sa_handler* member.

The *si_signo* member contains the system-generated signal number.

The *si_errno* member may contain implementation-defined additional error information; if non-zero, it contains an error number identifying the condition that caused the signal to be generated.

The *si_code* member contains a code identifying the cause of the signal.

If the value of *si_code* is less than or equal to 0, then the signal was generated by a process and *si_pid* and *si_uid*, respectively, indicate the process ID and the real user ID of the sender. The **<signal.h>** header description contains information about the signal specific contents of the elements of the **siginfo_t** type.

SA_NOCLDWAIT

If set, and *sig* equals SIGCHLD, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait*( ), *waitid*( ), and *waitpid*( ) shall fail and set *errno* to [ECHILD]. Otherwise, terminating child processes shall be transformed into zombie processes, unless SIGCHLD is set to SIG_IGN.

SA_NODEFER

If set and *sig* is caught, *sig* shall not be added to the process' signal mask on entry to the signal handler unless it is included in *sa_mask*. Otherwise, *sig* shall always be added to the process' signal mask on entry to the signal handler.

When a signal is caught by a signal-catching function installed by *sigaction*( ), a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either *sigprocmask*( ) or *sigsuspend*( ) is made). This mask is formed by taking the union of the current signal mask and the value of the *sa_mask* for the signal being delivered unless SA_NODEFER or SA_RESETHAND is set, and then including the signal being delivered. If and when the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it shall remain installed until another action is explicitly requested (by another call to *sigaction*( )), until the SA_RESETHAND flag causes resetting of the handler, or until one of the *exec* functions is called.

If the previous action for *sig* had been established by *signal*( ), the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular *oact*->*sa_handler* is not necessarily the same value passed to *signal*( ). However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to *sigaction*( ) via the *act* argument, handling of the signal shall be as if the original call to *signal*( ) were repeated.

If *sigaction*( ) fails, no new signal handler is installed.

It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to SIG_DFL is ignored or causes an error to be returned with *errno* set to [EINVAL].

If SA_SIGINFO is not set in *sa_flags*, then the disposition of subsequent occurrences of *sig* when it is already pending is implementation-defined; the signal-catching function shall be invoked with a single argument. If the implementation supports the Realtime Signals Extension option, and if SA_SIGINFO is set in *sa_flags*, then subsequent occurrences of *sig* generated by *sigqueue*( ) or as a result of any signal-generating function that supports the specification of an application-defined value (when *sig* is already pending) shall be queued in FIFO order until delivered or accepted; the signal-catching function shall be invoked with three arguments. The application specified value is passed to the signal-catching function as the *si_value* member of the **siginfo_t** structure.

The result of the use of *sigaction*( ) and a *sigwait*( ) function concurrently within a process on the same signal is unspecified.

RETURN VALUE

Upon successful completion, *sigaction*( ) shall return 0; otherwise, −1 shall be returned, *errno* shall be set to indicate the error, and no new signal-catching function shall be installed.

ERRORS

The *sigaction*( ) function shall fail if:

[EINVAL]	The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.
[ENOTSUP]	The SA_SIGINFO bit flag is set in the *sa_flags* field of the **sigaction** structure, and the implementation does not support either the Realtime Signals Extension option, or the X/Open System Interfaces Extension option.

The *sigaction*( ) function may fail if:

[EINVAL]	An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

EXAMPLES

None.

APPLICATION USAGE

The *sigaction*( ) function supersedes the *signal*( ) function, and should be used in preference. In particular, *sigaction*( ) and *signal*( ) should not be used in the same process to control the same signal. The behavior of reentrant functions, as defined in the DESCRIPTION, is as specified by this volume of IEEE Std 1003.1-200x, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restrictions. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep*( ) and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore (see *semget*( ), *sem_init*( ), *sem_open*( ), and so on). Note in particular that even the ‘‘safe’’ functions may modify *errno*; the signal-catching function, if not executing as an independent thread, may want to save and restore its value. Naturally, the same principles apply to the reentrancy of application routines and asynchronous data access. Note that *longjmp*( ) and *siglongjmp*( ) are not in the list of reentrant functions. This is because the code executing after *longjmp*( ) and *siglongjmp*( ) can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*( ) and *siglongjmp*( ) from within signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either *malloc*( ) or *free*( ) functions or the standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Since any combination of different functions using a common data structure can cause reentrancy problems, this volume of IEEE Std 1003.1-200x does not define the behavior when any unsafe function is called in a signal handler that interrupts an unsafe function.

If the signal occurs other than as the result of calling *abort*( ), *kill*( ), or *raise*( ), the behavior is undefined if the signal handler calls any function in the standard library other than one of the functions listed in the table above or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call fails, the value of *errno* is unspecified.

Usually, the signal is executed on the stack that was in effect before the signal was delivered. An alternate stack may be specified to receive a subset of the signals being caught.

When the signal handler returns, the receiving process resumes execution at the point it was interrupted unless the signal handler makes other arrangements. If *longjmp*( ) or *_longjmp*( ) is used to leave the signal handler, then the signal mask must be explicitly restored by the process.

This volume of IEEE Std 1003.1-200x defines the third argument of a signal handling function when SA_SIGINFO is set as a **void *** instead of a **ucontext_t ***, but without requiring type checking. New applications should explicitly cast the third argument of the signal handling function to **ucontext_t ***.

The BSD optional four argument signal handling function is not supported by this volume of IEEE Std 1003.1-200x. The BSD declaration would be:

```
void handler(int sig, int code, struct sigcontext *scp,
    char *addr);
```

where *sig* is the signal number, *code* is additional information on certain signals, *scp* is a pointer to the sigcontext structure, and *addr* is additional address information. Much the same information is available in the objects pointed to by the second argument of the signal handler specified when SA_SIGINFO is set.

RATIONALE

Although this volume of IEEE Std 1003.1-200x requires that signals that cannot be ignored shall not be added to the signal mask when a signal-catching function is entered, there is no explicit requirement that subsequent calls to *sigaction*( ) reflect this in the information returned in the *oact* argument. In other words, if SIGKILL is included in the *sa_mask* field of *act*, it is unspecified whether or not a subsequent call to *sigaction*( ) returns with SIGKILL included in the *sa_mask* field of *oact*.

The SA_NOCLDSTOP flag, when supplied in the *act*->*sa_flags* parameter, allows overloading SIGCHLD with the System V semantics that each SIGCLD signal indicates a single terminated child. Most conforming applications that catch SIGCHLD are expected to install signal-catching functions that repeatedly call the *waitpid*( ) function with the WNOHANG flag set, acting on each child for which status is returned, until *waitpid*( ) returns zero. If stopped children are not of interest, the use of the SA_NOCLDSTOP flag can prevent the overhead from invoking the signal-catching routine when they stop.

Some historical implementations also define other mechanisms for stopping processes, such as the *ptrace*( ) function. These implementations usually do not generate a SIGCHLD signal when processes stop due to this mechanism; however, that is beyond the scope of this volume of IEEE Std 1003.1-200x.

This volume of IEEE Std 1003.1-200x requires that calls to *sigaction*( ) that supply a NULL *act* argument succeed, even in the case of signals that cannot be caught or ignored (that is, SIGKILL or SIGSTOP). The System V *signal*( ) and BSD *sigvec*( ) functions return [EINVAL] in these cases and, in this respect, their behavior varies from *sigaction*( ).

This volume of IEEE Std 1003.1-200x requires that *sigaction*( ) properly save and restore a signal action set up by the ISO C standard *signal*( ) function. However, there is no guarantee that the reverse is true, nor could there be given the greater amount of information conveyed by the **sigaction** structure. Because of this, applications should avoid using both functions for the same signal in the same process. Since this cannot always be avoided in case of general-purpose library routines, they should always be implemented with *sigaction*( ).

It was intended that the *signal*( ) function should be implementable as a library routine using *sigaction*( ).

The POSIX Realtime Extension extends the *sigaction*( ) function as specified by the POSIX.1-1990 standard to allow the application to request on a per-signal basis via an additional signal action flag that the extra parameters, including the application-defined signal value, if any, be passed to the signal-catching function.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *bsd_signal*( ), *kill*( ), *_longjmp*( ), *longjmp*( ), *raise*( ), *semget*( ), *sem_init*( ), *sem_open*( ), *sigaddset*( ), *sigaltstack*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *signal*( ), *sigprocmask*( ), *sigsuspend*( ), *wait*( ), *waitid*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<ucontext.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and POSIX Threads Extension.

In the DESCRIPTION, the second argument to *func* when SA_SIGINFO is set is no longer permitted to be NULL, and the description of permitted **siginfo_t** contents is expanded by reference to **<signal.h>**.

Since the X/OPEN UNIX Extension functionality is now folded into the BASE, the [ENOTSUP] error is deleted.

Issue 6

The Open Group Corrigendum U028/7 is applied. In the paragraph entitled ‘‘Signal Effects on Other Functions’’, a reference to *sigpending*( ) is added.

In the DESCRIPTION, the text ‘‘Signal Generation and Delivery’’, ‘‘Signal Actions’’, and ‘‘Signal Effects on Other Functions’’ are moved to a separate section of this volume of IEEE Std 1003.1-200x.

Text describing functionality from the Realtime Signals option is marked.

The following changes are made for alignment with the ISO POSIX-1: 1996 standard:

- The [ENOTSUP] error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *sigaction*( ) prototype for alignment with the ISO/IEC 9899: 1999 standard.

References to the *wait3*( ) function are removed.

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sigaddset — add a signal to a signal set

SYNOPSIS

CX
```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
```

DESCRIPTION

The *sigaddset*( ) function adds the individual signal specified by the *signo* to the signal set pointed to by *set*.

Applications shall call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *pthread_sigmask*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigtimedwait*( ), *sigwait*( ), or *sigwaitinfo*( ), the results are undefined.

RETURN VALUE

Upon successful completion, *sigaddset*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *sigaddset*( ) function may fail if:

[EINVAL]	The value of the *signo* argument is an invalid or unsupported signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sigaltstack — set and get signal alternate stack context

SYNOPSIS

XSI

```
#include <signal.h>

int sigaltstack(const stack_t *restrict ss, stack_t *restrict oss);
```

DESCRIPTION

The *sigaltstack*( ) function allows a process to define and examine the state of an alternate stack for signal handlers. Signals that have been explicitly declared to execute on the alternate stack shall be delivered on the alternate stack.

If *ss* is not a null pointer, it points to a **stack_t** structure that specifies the alternate signal stack that shall take effect upon return from *sigaltstack*( ). The *ss_flags* member specifies the new stack state. If it is set to SS_DISABLE, the stack is disabled and *ss_sp* and *ss_size* are ignored. Otherwise, the stack shall be enabled, and the *ss_sp* and *ss_size* members specify the new address and size of the stack.

The range of addresses starting at *ss_sp* up to but not including *ss_sp*+*ss_size*, is available to the implementation for use as the stack. This function makes no assumptions regarding which end is the stack base and in which direction the stack grows as items are pushed.

If *oss* is not a null pointer, on successful completion it shall point to a **stack_t** structure that specifies the alternate signal stack that was in effect prior to the call to *sigaltstack*( ). The *ss_sp* and *ss_size* members specify the address and size of that stack. The *ss_flags* member specifies the stack's state, and may contain one of the following values:

SS_ONSTACK	The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing on it fail. This flag shall not be modified by processes.
SS_DISABLE	The alternate signal stack is currently disabled.

The value SIGSTKSZ is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value MINSIGSTKSZ is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead. The constants SS_ONSTACK, SS_DISABLE, SIGSTKSZ, and MINSIGSTKSZ are defined in **<signal.h>**.

After a successful call to one of the *exec* functions, there are no alternate signal stacks in the new process image.

In some implementations, a signal (whether or not indicated to execute on the alternate stack) shall always execute on the alternate stack if it is delivered while another signal is being caught using the alternate stack.

Use of this function by library threads that are not bound to kernel-scheduled entities results in undefined behavior.

RETURN VALUE

Upon successful completion, *sigaltstack*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *sigaltstack*( ) function shall fail if:

[EINVAL]	The *ss* argument is not a null pointer, and the *ss_flags* member pointed to by *ss* contains flags other than SS_DISABLE.
[ENOMEM]	The size of the alternate stack area is less than MINSIGSTKSZ.
[EPERM]	An attempt was made to modify an active stack.

EXAMPLES

Allocating Memory for an Alternate Stack

The following example illustrates a method for allocating memory for an alternate stack.

```
#include <signal.h>
...
if ((sigstk.ss_sp = malloc(SIGSTKSZ)) == NULL)
    /* Error return. */
sigstk.ss_size = SIGSTKSZ;
sigstk.ss_flags = 0;
if (sigaltstack(&sigstk,(stack_t *)0) < 0)
    perror("sigaltstack");
```

APPLICATION USAGE

On some implementations, stack space is automatically extended as needed. On those implementations, automatic extension is typically not available for an alternate stack. If the stack overflows, the behavior is undefined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), *sigsetjmp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The last sentence of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *sigaltstack*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

sigdelset — delete a signal from a signal set

SYNOPSIS

CX
```
#include <signal.h>

int sigdelset(sigset_t *set, int signo);
```

DESCRIPTION

The *sigdelset*( ) function deletes the individual signal specified by *signo* from the signal set pointed to by *set*.

Applications should call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *pthread_sigmask*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigtimedwait*( ), *sigwait*( ), or *sigwaitinfo*( ), the results are undefined.

RETURN VALUE

Upon successful completion, *sigdelset*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *sigdelset*( ) function may fail if:

[EINVAL]	The *signo* argument is not a valid signal number, or is an unsupported signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sigemptyset — initialize and empty a signal set

SYNOPSIS

CX

```
#include <signal.h>

int sigemptyset(sigset_t *set);
```

DESCRIPTION

The *sigemptyset*( ) function initializes the signal set pointed to by *set*, such that all signals defined in IEEE Std 1003.1-200x are excluded.

RETURN VALUE

Upon successful completion, *sigemptyset*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The implementation of the *sigemptyset*( ) (or *sigfillset*( )) function could quite trivially clear (or set) all the bits in the signal set. Alternatively, it would be reasonable to initialize part of the structure, such as a version field, to permit binary-compatibility between releases where the size of the set varies. For such reasons, either *sigemptyset*( ) or *sigfillset*( ) must be called prior to any other use of the signal set, even if such use is read-only (for example, as an argument to *sigpending*( )). This function is not intended for dynamic allocation.

The *sigfillset*( ) and *sigemptyset*( ) functions require that the resulting signal set include (or exclude) all the signals defined in this volume of IEEE Std 1003.1-200x. Although it is outside the scope of this volume of IEEE Std 1003.1-200x to place this requirement on signals that are implemented as extensions, it is recommended that implementation-defined signals also be affected by these functions. However, there may be a good reason for a particular signal not to be affected. For example, blocking or ignoring an implementation-defined signal may have undesirable side effects, whereas the default action for that signal is harmless. In such a case, it would be preferable for such a signal to be excluded from the signal set returned by *sigfillset*( ).

In early proposals there was no distinction between invalid and unsupported signals (the names of optional signals that were not supported by an implementation were not defined by that implementation). The [EINVAL] error was thus specified as a required error for invalid signals. With that distinction, it is not necessary to require implementations of these functions to determine whether an optional signal is actually supported, as that could have a significant performance impact for little value. The error could have been required for invalid signals and optional for unsupported signals, but this seemed unnecessarily complex. Thus, the error is optional in both cases.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigfillset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sigfillset — initialize and fill a signal set

SYNOPSIS

CX
```
#include <signal.h>

int sigfillset(sigset_t *set);
```

DESCRIPTION

The *sigfillset*( ) function shall initialize the signal set pointed to by *set*, such that all signals defined in this volume of IEEE Std 1003.1-200x are included.

RETURN VALUE

Upon successful completion, *sigfillset*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to *sigemptyset*( ) (on page 1848).

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sighold, sigignore, sigpause, sigrelse, sigset — signal management

SYNOPSIS

XSI

```
#include <signal.h>

int sighold(int sig);
int sigignore(int sig);
int sigpause(int sig);
int sigrelse(int sig);
void (*sigset(int sig, void (*disp)(int)))(int);

```

DESCRIPTION

Use of any of these functions is unspecified in a multi-threaded process.

The *sighold*( ), *sigignore*( ), *sigpause*( ), *sigrelse*( ), and *sigset*( ) functions provide simplified signal management.

The *sigset*( ) function shall modify signal dispositions. The *sig* argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The *disp* argument specifies the signal's disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If *sigset*( ) is used, and *disp* is the address of a signal handler, the system shall add *sig* to the calling process' signal mask before executing the signal handler; when the signal handler returns, the system shall restore the calling process' signal mask to its state prior to the delivery of the signal. In addition, if *sigset*( ) is used, and *disp* is equal to SIG_HOLD, *sig* shall be added to the calling process' signal mask and *sig*'s disposition shall remain unchanged. If *sigset*( ) is used, and *disp* is not equal to SIG_HOLD, *sig* shall be removed from the calling process' signal mask.

The *sighold*( ) function shall add *sig* to the calling process' signal mask.

The *sigrelse*( ) function shall remove *sig* from the calling process' signal mask.

The *sigignore*( ) function shall set the disposition of *sig* to SIG_IGN.

The *sigpause*( ) function shall remove *sig* from the calling process' signal mask and suspend the calling process until a signal is received. The *sigpause*( ) function shall restore the process' signal mask to its original state before returning.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes shall not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it shall block until all of its children terminate, and *wait*( ), *waitid*( ), and *waitpid*( ) shall fail and set *errno* to [ECHILD].

RETURN VALUE

Upon successful completion, *sigset*( ) shall return SIG_HOLD if the signal had been blocked and the signal's previous disposition if it had not been blocked. Otherwise, SIG_ERR shall be returned and *errno* set to indicate the error.

The *sigpause*( ) function shall suspend execution of the thread until a signal is received, whereupon it shall return −1 and set *errno* to [EINTR].

For all other functions, upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

These functions shall fail if:

[EINVAL] The *sig* argument is an illegal signal number.

The *sigset*( ) and *sigignore*( ) functions shall fail if:

[EINVAL] An attempt is made to catch a signal that cannot be caught, or to ignore a signal that cannot be ignored.

EXAMPLES

None.

APPLICATION USAGE

The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use *sigaction*( ) rather than *sigset*( ).

The *sighold*( ) function, in conjunction with *sigrelse*( ) or *sigpause*( ), may be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

The *sigsuspend*( ) function should be used in preference to *sigpause*( ) for broader portability.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *exec*, *pause*( ), *sigaction*( ), *signal*( ), *sigsuspend*( ), *waitid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4 Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the *sigpause*( ) function restores the process' signal mask to its original state before returning.

The RETURN VALUE section is updated to indicate that the *sigpause*( ) function suspends execution of the process until a signal is received, whereupon it returns –1 and sets *errno* to [EINTR].

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

References to the *wait3*( ) function are removed.

The XSI functions are split out into their own reference page.

NAME

sigignore — signal management

SYNOPSIS

XSI

```
#include <signal.h>

int sigignore(int sig);
```

DESCRIPTION

Refer to *sighold*( ).

NAME

siginterrupt — allow signals to interrupt functions

SYNOPSIS

XSI

```
#include <signal.h>

int siginterrupt(int sig, int flag);
```

DESCRIPTION

The *siginterrupt*( ) function shall change the restart behavior when a function is interrupted by the specified signal. The function *siginterrupt*(*sig*, *flag*) has an effect as if implemented as: |

```
siginterrupt(int sig, int flag) {
    int ret;
    struct sigaction act;

    (void) sigaction(sig, NULL, &act);
    if (flag)
        act.sa_flags &= ˜SA_RESTART;
    else
        act.sa_flags |= SA_RESTART;
    ret = sigaction(sig, &act, NULL);
    return ret;
}
```

RETURN VALUE

Upon successful completion, *siginterrupt*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *siginterrupt*( ) function shall fail if:

[EINVAL]	The *sig* argument is not a valid signal number.

EXAMPLES

None.

APPLICATION USAGE

The *siginterrupt*( ) function supports programs written to historical system interfaces. A conforming application, when being written or rewritten, should use *sigaction*( ) with the SA_RESTART flag instead of *siginterrupt*( ). |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

sigismember — test for a signal in a signal set

SYNOPSIS

CX

```
#include <signal.h>

int sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

The *sigismember*( ) function shall test whether the signal specified by *signo* is a member of the set pointed to by *set*.

Applications should call either *sigemptyset*( ) or *sigfillset*( ) at least once for each object of type **sigset_t** prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of *pthread_sigmask*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigismember*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigtimedwait*( ), *sigwait*( ), or *sigwaitinfo*( ), the results are undefined.

RETURN VALUE

Upon successful completion, *sigismember*( ) shall return 1 if the specified signal is a member of the specified set, or 0 if it is not. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *sigismember*( ) function may fail if:

[EINVAL] The *signo* argument is not a valid signal number, or is an unsupported signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigfillset*( ), *sigemptyset*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

siglongjmp — non-local goto with signal handling

SYNOPSIS

CX
```
#include <setjmp.h>

void siglongjmp(sigjmp_buf env, int val);
```

DESCRIPTION

The *siglongjmp*( ) function shall be equivalent to the *longjmp*( ) function, except as follows:

- References to *setjmp*( ) shall be equivalent to *sigsetjmp*( ).
- The *siglongjmp*( ) function shall restore the saved signal mask if and only if the *env* argument was initialized by a call to *sigsetjmp*( ) with a non-zero *savemask* argument.

RETURN VALUE

After *siglongjmp*( ) is completed, program execution shall continue as if the corresponding invocation of *sigsetjmp*( ) had just returned the value specified by *val*. The *siglongjmp*( ) function shall not cause *sigsetjmp*( ) to return 0; if *val* is 0, *sigsetjmp*( ) shall return the value 1.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The distinction between *setjmp*( ) or *longjmp*( ) and *sigsetjmp*( ) or *siglongjmp*( ) is only significant for programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

longjmp( ), *setjmp*( ), *sigprocmask*( ), *sigsetjmp*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ISO POSIX-1 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is rewritten in terms of *longjmp*( ).

The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an extension over the ISO C standard.

NAME

signal — signal management

SYNOPSIS

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX Use of this function is unspecified in a multi-threaded process.

The *signal*( ) function chooses one of three ways in which receipt of the signal number *sig* is to be subsequently handled. If the value of *func* is SIG_DFL, default handling for that signal shall occur. If the value of *func* is SIG_IGN, the signal shall be ignored. Otherwise, the application shall ensure that *func* points to a function to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than functions in the standard library), is called a ‘‘signal handler’’.

When a signal occurs, and *func* points to a function, it is implementation-defined whether the equivalent of a:

```
signal(sig, SIG_DFL);
```

is executed or the implementation prevents some implementation-defined set of signals (at least including *sig*) from occurring until the current signal handling has completed. (If the value of *sig* is SIGILL, the implementation may alternatively define that no action is taken.) Next the equivalent of:

```
(*func)(sig);
```

is executed. If and when the function returns, if the value of *sig* was SIGFPE, SIGILL, or SIGSEGV or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program shall resume execution at the point it was interrupted. CX If the signal occurs as the result of calling the *abort*( ), *raise*( ), *kill*( ), *pthread_kill*( ), or *sigqueue*( ) function, the signal handler shall not call the *raise*( ) function. |

CX If the signal occurs other than as the result of calling *abort*( ), *raise*( ), *kill*( ), *pthread_kill*( ), or *sigqueue*( ), the behavior is undefined if the signal handler refers to any object with static storage duration other than by assigning a value to an object declared as volatile **sig_atomic_t**, or if the signal handler calls any function in the standard library other than one of the functions listed in Section 2.4 (on page 478). Furthermore, if such a call fails, the value of *errno* is unspecified. |

At program start-up, the equivalent of:

```
signal(sig, SIG_IGN);
```

is executed for some signals, and the equivalent of:

```
signal(sig, SIG_DFL);
```

CX is executed for all other signals (see *exec*).

RETURN VALUE

If the request can be honored, *signal*( ) shall return the value of *func* for the most recent call to *signal*( ) for the specified signal *sig*. Otherwise, SIG_ERR shall be returned and a positive value shall be stored in *errno*.

ERRORS

The *signal*( ) function shall fail if:

CX [EINVAL] The *sig* argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

The *signal*( ) function may fail if:

CX [EINVAL] An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

EXAMPLES

None.

APPLICATION USAGE

The *sigaction*( ) function provides a more comprehensive and reliable mechanism for controlling signals; new applications should use *sigaction*( ) rather than *signal*( ).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *exec*, *pause*( ), *sigaction*( ), *sigsuspend*( ), *waitid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated to indicate that the *sigpause*( ) function restores the process' signal mask to its original state before returning.

The RETURN VALUE section is updated to indicate that the *sigpause*( ) function suspends execution of the process until a signal is received, whereupon it returns –1 and sets *errno* to [EINTR].

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

References to the *wait3*( ) function are removed.

The *sighold*( ), *sigignore*( ), *sigrelse*( ), and *sigset*( ) functions are split out onto their own reference page.

NAME

signbit — test sign

SYNOPSIS

```
#include <math.h>

int signbit(real-floating x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *signbit*( ) macro shall determine whether the sign of its argument value is negative. NaNs, zeros, and infinities have a sign bit.

RETURN VALUE

The *signbit*( ) macro shall return a non-zero value if and only if the sign of its argument value is negative.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fpclassify( ), *isfinite*( ), *isinf*( ), *isnan*( ), *isnormal*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

sigpause — remove a signal from the signal mask and suspend the thread

SYNOPSIS

XSI
```
#include <signal.h>

int sigpause(int sig);
```

DESCRIPTION

Refer to *sighold*( ).

NAME

sigpending — examine pending signals

SYNOPSIS

CX

```
#include <signal.h>

int sigpending(sigset_t *set);
```

DESCRIPTION

The *sigpending*( ) function shall store, in the location referenced by the *set* argument, the set of signals that are blocked from delivery to the calling thread and that are pending on the process or the calling thread.

RETURN VALUE

Upon successful completion, *sigpending*( ) shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sigaddset( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *sigprocmask*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sigprocmask — examine and change blocked signals

SYNOPSIS

CX
```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
    sigset_t *restrict oset);
```

DESCRIPTION

Refer to *pthread_sigmask*( ).

NAME

sigqueue — queue a signal to a process (**REALTIME**)

SYNOPSIS

RTS

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```

DESCRIPTION

The *sigqueue*( ) function shall cause the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*.

The conditions required for a process to have permission to queue a signal to another process are the same as for the *kill*( ) function.

The *sigqueue*( ) function shall return immediately. If SA_SIGINFO is set for *signo* and if the resources were available to queue the signal, the signal shall be queued and sent to the receiving process. If SA_SIGINFO is not set for *signo*, then *signo* shall be sent at least once to the receiving process; it is unspecified whether *value* shall be sent to the receiving process as a result of this call.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a *sigwait*( ) function for *signo*, either *signo* or at least the pending, unblocked signal shall be delivered to the calling thread before the *sigqueue*( ) function returns. Should any multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected for delivery, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

RETURN VALUE

Upon successful completion, the specified signal shall have been queued, and the *sigqueue*( ) function shall return a value of zero. Otherwise, the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *sigqueue*( ) function shall fail if:

[EAGAIN]	No resources available to queue the signal. The process has already queued SIGQUEUE_MAX signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.
[EINVAL]	The value of the *signo* argument is an invalid or unsupported signal number.
[EPERM]	The process does not have the appropriate privilege to send the signal to the receiving process.
[ESRCH]	The process *pid* does not exist.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *sigqueue*( ) function allows an application to queue a realtime signal to itself or to another process, specifying the application-defined value. This is common practice in realtime applications on existing realtime systems. It was felt that specifying another function in the *sig*... name space already carved out for signals was preferable to extending the interface to *kill*( ).

Such a function became necessary when the put/get event function of the message queues was removed. It should be noted that the *sigqueue*( ) function implies reduced performance in a security-conscious implementation as the access permissions between the sender and receiver have to be checked on each send when the *pid* is resolved into a target process. Such access checks were necessary only at message queue open in the previous interface.

The standard developers required that *sigqueue*( ) have the same semantics with respect to the null signal as *kill*( ), and that the same permission checking be used. But because of the difficulty of implementing the ‘‘broadcast’’ semantic of *kill*( ) (for example, to process groups) and the interaction with resource allocation, this semantic was not adopted. The *sigqueue*( ) function queues a signal to a single process specified by the *pid* argument.

The *sigqueue*( ) function can fail if the system has insufficient resources to queue the signal. An explicit limit on the number of queued signals that a process could send was introduced. While the limit is ‘‘per-sender’’, this volume of IEEE Std 1003.1-200x does not specify that the resources be part of the state of the sender. This would require either that the sender be maintained after exit until all signals that it had sent to other processes were handled or that all such signals that had not yet been acted upon be removed from the queue(s) of the receivers. This volume of IEEE Std 1003.1-200x does not preclude this behavior, but an implementation that allocated queuing resources from a system-wide pool (with per-sender limits) and that leaves queued signals pending after the sender exits is also permitted.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 491), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The *sigqueue*( ) function is marked as part of the Realtime Signals Extension option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

NAME

sigrelse — remove a signal from signal mask or modify signal disposition

SYNOPSIS

XSI

```
#include <signal.h>

int sigrelse(int sig);
```

DESCRIPTION

Refer to *sighold*( ).

NAME

sigset — signal management

SYNOPSIS

```
#include <signal.h>
```

XSI
```
void (*sigset(int sig, void (*disp)(int)))(int);
```

DESCRIPTION

Refer to *sighold*( ).

NAME

sigsetjmp — set jump point for a non-local goto

SYNOPSIS

CX
```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

DESCRIPTION

The *sigsetjmp*( ) function shall be equivalent to the *setjmp*( ) function, except as follows:

- References to *setjmp*( ) are equivalent to *sigsetjmp*( ).
- References to *longjmp*( ) are equivalent to *siglongjmp*( ).
- If the value of the *savemask* argument is not 0, *sigsetjmp*( ) shall also save the current signal mask of the calling thread as part of the calling environment.

RETURN VALUE

If the return is from a successful direct invocation, *sigsetjmp*( ) shall return 0. If the return is from a call to *siglongjmp*( ), *sigsetjmp*( ) shall return a non-zero value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The distinction between *setjmp*( )/*longjmp*( ) and *sigsetjmp*( )/*siglongjmp*( ) is only significant for programs which use *sigaction*( ), *sigprocmask*( ), or *sigsuspend*( ).

Note that since this function is defined in terms of *setjmp*( ), if *savemask* is zero, it is unspecified whether the signal mask is saved.

RATIONALE

The ISO C standard specifies various restrictions on the usage of the *setjmp*( ) macro in order to permit implementors to recognize the name in the compiler and not implement an actual function. These same restrictions apply to the *sigsetjmp*( ) macro.

There are processors that cannot easily support these calls, but this was not considered a sufficient reason to exclude them.

4.2 BSD, 4.3 BSD, and XSI-conformant systems provide functions named *_setjmp*( ) and *_longjmp*( ) that, together with *setjmp*( ) and *longjmp*( ), provide the same functionality as *sigsetjmp*( ) and *siglongjmp*( ). On those systems, *setjmp*( ) and *longjmp*( ) save and restore signal masks, while *_setjmp*( ) and *_longjmp*( ) do not. On System V, Release 3 and in corresponding issues of the SVID, *setjmp*( ) and *longjmp*( ) are explicitly defined not to save and restore signal masks. In order to permit existing practice in both cases, the relation of *setjmp*( ) and *longjmp*( ) to signal masks is not specified, and a new set of functions is defined instead.

The *longjmp*( ) and *siglongjmp*( ) functions operate as in the previous issue provided the matching *setjmp*( ) or *sigsetjmp*( ) has been performed in the same thread. Non-local jumps into contexts saved by other threads would be at best a questionable practice and were not considered worthy of standardization.

FUTURE DIRECTIONS

None.

SEE ALSO

siglongjmp( ), *signal*( ), *sigprocmask*( ), *sigsuspend*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<setjmp.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is reworded in terms of *setjmp*( ).

The SYNOPSIS is marked CX since the presence of this function in the **<setjmp.h>** header is an extension over the ISO C standard.

NAME

sigsuspend — wait for a signal

SYNOPSIS

CX
```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

DESCRIPTION

The *sigsuspend*( ) function shall replace the current signal mask of the calling thread with the set of signals pointed to by *sigmask* and then suspend the thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. This shall not cause any other signals that may have been pending on the process to become pending on the thread.

If the action is to terminate the process then *sigsuspend*( ) shall never return. If the action is to execute a signal-catching function, then *sigsuspend*( ) shall return after the signal-catching function returns, with the signal mask restored to the set that existed prior to the *sigsuspend*( ) call.

It is not possible to block signals that cannot be ignored. This is enforced by the system without causing an error to be indicated.

RETURN VALUE

Since *sigsuspend*( ) suspends thread execution indefinitely, there is no successful completion return value. If a return occurs, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *sigsuspend*( ) function shall fail if:

[EINTR] A signal is caught by the calling process and control is returned from the signal-catching function.

EXAMPLES

None.

APPLICATION USAGE

Normally, at the beginning of a critical code section, a specified set of signals is blocked using the *sigprocmask*( ) function. When the thread has completed the critical section and needs to wait for the previously blocked signal(s), it pauses by calling *sigsuspend*( ) with the mask that was returned by the *sigprocmask*( ) call.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), *pause*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The text in the RETURN VALUE section has been changed from ‘‘suspends process execution’’ to ‘‘suspends thread execution’’. This reflects IEEE PASC Interpretation 1003.1c #40.

Text in the APPLICATION USAGE section has been replaced.

The SYNOPSIS is marked CX since the presence of this function in the **<signal.h>** header is an extension over the ISO C standard.

NAME

sigtimedwait, sigwaitinfo — wait for queued signals (**REALTIME**)

SYNOPSIS

RTS

```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
    siginfo_t *restrict info,
    const struct timespec *restrict timeout);
int sigwaitinfo(const sigset_t *restrict set,
    siginfo_t *restrict info);
```

DESCRIPTION

The *sigtimedwait*( ) function shall be equivalent to *sigwaitinfo*( ) except that if none of the signals specified by *set* are pending, *sigtimedwait*( ) shall wait for the time interval specified in the **timespec** structure referenced by *timeout*. If the **timespec** structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then *sigtimedwait*( ) shall return immediately with an error. If *timeout* is the NULL pointer, the behavior is unspecified. MON If the Monotonic Clock option is supported, the CLOCK_MONOTONIC clock shall be used to measure the time interval specified by the *timeout* argument.

The *sigwaitinfo*( ) function selects the pending signal from the set specified by *set*. Should any of multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified. If no signal in *set* is pending at the time of the call, the calling thread shall be suspended until one or more signals in *set* become pending or until it is interrupted by an unblocked, caught signal.

The *sigwaitinfo*( ) function shall be equivalent to the *sigwait*( ) function if the *info* argument is NULL. If the *info* argument is non-NULL, the *sigwaitinfo*( ) function shall be equivalent to *sigwait*( ), except that the selected signal number shall be stored in the *si_signo* member, and the cause of the signal shall be stored in the *si_code* member. If any value is queued to the selected signal, the first such queued value shall be dequeued and, if the *info* argument is non-NULL, the value shall be stored in the *si_value* member of *info*. The system resource used to queue the signal shall be released and returned to the system for other use. If no value is queued, the content of the *si_value* member is undefined. If no further signals are queued for the selected signal, the pending indication for that signal shall be reset.

RETURN VALUE

Upon successful completion (that is, one of the signals specified by *set* is pending or is generated) *sigwaitinfo*( ) and *sigtimedwait*( ) shall return the selected signal number. Otherwise, the function shall return a value of –1 and set *errno* to indicate the error.

ERRORS

The *sigtimedwait*( ) function shall fail if:

[EAGAIN] No signal specified by *set* was generated within the specified timeout period.

The *sigtimedwait*( ) and *sigwaitinfo*( ) functions may fail if:

[EINTR] The wait was interrupted by an unblocked, caught signal. It shall be documented in system documentation whether this error causes these functions to fail.

The *sigtimedwait*( ) function may also fail if:

[EINVAL] The *timeout* argument specified a *tv_nsec* value less than zero or greater than or equal to 1 000 million.

An implementation only checks for this error if no signal is pending in *set* and it is necessary to wait.

EXAMPLES

None.

APPLICATION USAGE

The *sigtimedwait*( ) function times out and returns an [EAGAIN] error. Application writers should note that this is inconsistent with other functions such as *pthread_cond_timedwait*( ) that return [ETIMEDOUT].

RATIONALE

Existing programming practice on realtime systems uses the ability to pause waiting for a selected set of events and handle the first event that occurs in-line instead of in a signal-handling function. This allows applications to be written in an event-directed style similar to a state machine. This style of programming is useful for largescale transaction processing in which the overall throughput of an application and the ability to clearly track states are more important than the ability to minimize the response time of individual event handling.

It is possible to construct a signal-waiting macro function out of the realtime signal function mechanism defined in this volume of IEEE Std 1003.1-200x. However, such a macro has to include the definition of a generalized handler for all signals to be waited on. A significant portion of the overhead of handler processing can be avoided if the signal-waiting function is provided by the kernel. This volume of IEEE Std 1003.1-200x therefore provides two signal-waiting functions—one that waits indefinitely and one with a timeout—as part of the overall realtime signal function specification.

The specification of a function with a timeout allows an application to be written that can be broken out of a wait after a set period of time if no event has occurred. It was argued that setting a timer event before the wait and recognizing the timer event in the wait would also implement the same functionality, but at a lower performance level. Because of the performance degradation associated with the user-level specification of a timer event and the subsequent cancelation of that timer event after the wait completes for a valid event, and the complexity associated with handling potential race conditions associated with the user-level method, the separate function has been included.

Note that the semantics of the *sigwaitinfo*( ) function are nearly identical to that of the *sigwait*( ) function defined by this volume of IEEE Std 1003.1-200x. The only difference is that *sigwaitinfo*( ) returns the queued signal value in the *value* argument. The return of the queued value is required so that applications can differentiate between multiple events queued to the same signal number.

The two distinct functions are being maintained because some implementations may choose to implement the POSIX Threads Extension functions and not implement the queued signals extensions. Note, though, that *sigwaitinfo*( ) does not return the queued value if the *value* argument is NULL, so the POSIX Threads Extension *sigwait*( ) function can be implemented as a macro on *sigwaitinfo*( ).

The *sigtimedwait*( ) function was separated from the *sigwaitinfo*( ) function to address concerns regarding the overloading of the *timeout* pointer to indicate indefinite wait (no timeout), timed wait, and immediate return, and concerns regarding consistency with other functions where the conditional and timed waits were separate functions from the pure blocking function. The semantics of *sigtimedwait*( ) are specified such that *sigwaitinfo*( ) could be implemented as a macro with a NULL pointer for *timeout*.

The *sigwait* functions provide a synchronous mechanism for threads to wait for asynchronously generated signals. One important question was how many threads that are suspended in a call to a *sigwait*( ) function for a signal should return from the call when the signal is sent. Four choices were considered:

1. Return an error for multiple simultaneous calls to *sigwait* functions for the same signal.
2. One or more threads return.
3. All waiting threads return.
4. Exactly one thread returns.

Prohibiting multiple calls to *sigwait*( ) for the same signal was felt to be overly restrictive. The ''one or more'' behavior made implementation of conforming packages easy at the expense of forcing POSIX threads clients to protect against multiple simultaneous calls to *sigwait*( ) in application code in order to achieve predictable behavior. There was concern that the ''all waiting threads'' behavior would result in ''signal broadcast storms'', consuming excessive CPU resources by replicating the signals in the general case. Furthermore, no convincing examples could be presented that delivery to all was either simpler or more powerful than delivery to one.

Thus, the consensus was that exactly one thread that was suspended in a call to a *sigwait* function for a signal should return when that signal occurs. This is not an onerous restriction as:

- A multi-way signal wait can be built from the single-way wait.
- Signals should only be handled by application-level code, as library routines cannot guess what the application wants to do with signals generated for the entire process.
- Applications can thus arrange for a single thread to wait for any given signal and call any needed routines upon its arrival.

In an application that is using signals for interprocess communication, signal processing is typically done in one place. Alternatively, if the signal is being caught so that process cleanup can be done, the signal handler thread can call separate process cleanup routines for each portion of the application. Since the application main line started each portion of the application, it is at the right abstraction level to tell each portion of the application to clean up.

Certainly, there exist programming styles where it is logical to consider waiting for a single signal in multiple threads. A simple *sigwait_multiple*( ) routine can be constructed to achieve this goal. A possible implementation would be to have each *sigwait_multiple*( ) caller registered as having expressed interest in a set of signals. The caller then waits on a thread-specific condition variable. A single server thread calls a *sigwait*( ) function on the union of all registered signals. When the *sigwait*( ) function returns, the appropriate state is set and condition variables are broadcast. New *sigwait_multiple*( ) callers may cause the pending *sigwait*( ) call to be canceled and reissued in order to update the set of signals being waited for.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.8.1 (on page 491), *pause*( ), *pthread_sigmask*( ), *sigaction*( ), *sigpending*( ), *sigsuspend*( ), *sigwait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

These functions are marked as part of the Realtime Signals Extension option.

The Open Group Corrigendum U035/3 is applied. The SYNOPSIS of the *sigwaitinfo*( ) function has been corrected so that the second argument is of type **siginfo_t ***.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Realtime Signals Extension option.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that the CLOCK_MONOTONIC clock, if supported, is used to measure timeout intervals.

The **restrict** keyword is added to the *sigtimedwait*( ) and *sigwaitinfo*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

NAME

sigwait — wait for queued signals

SYNOPSIS

CX

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict sig);
```

DESCRIPTION

The *sigwait*( ) function shall select a pending signal from *set*, atomically clear it from the system's set of pending signals, and return that signal number in the location referenced by *sig*. If prior to the call to *sigwait*( ) there are multiple pending instances of a single signal number, it is implementation-defined whether upon successful return there are any remaining pending signals for that signal number. RTS If the implementation supports queued signals and there are multiple signals queued for the signal number selected, the first such queued signal shall cause a return from *sigwait*( ) and the remainder shall remain queued. If no signal in *set* is pending at the time of the call, the thread shall be suspended until one or more becomes pending. The signals defined by *set* shall have been blocked at the time of the call to *sigwait*( ); otherwise, the behavior is undefined. The effect of *sigwait*( ) on the signal actions for the signals in *set* is unspecified.

If more than one thread is using *sigwait*( ) to wait for the same signal, no more than one of these threads shall return from *sigwait*( ) with the signal number. Which thread returns from *sigwait*( ) if more than a single thread is waiting is unspecified.

RTS Should any of the multiple pending signals in the range SIGRTMIN to SIGRTMAX be selected, it shall be the lowest numbered one. The selection order between realtime and non-realtime signals, or between multiple pending non-realtime signals, is unspecified.

RETURN VALUE

Upon successful completion, *sigwait*( ) shall store the signal number of the received signal at the location referenced by *sig* and return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *sigwait*( ) function may fail if:

[EINVAL]	The *set* argument contains an invalid or unsupported signal number.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

To provide a convenient way for a thread to wait for a signal, this volume of IEEE Std 1003.1-200x provides the *sigwait*( ) function. For most cases where a thread has to wait for a signal, the *sigwait*( ) function should be quite convenient, efficient, and adequate.

However, requests were made for a lower-level primitive than *sigwait*( ) and for semaphores that could be used by threads. After some consideration, threads were allowed to use semaphores and *sem_post*( ) was defined to be async-signal and async-cancel-safe.

In summary, when it is necessary for code run in response to an asynchronous signal to notify a thread, *sigwait*( ) should be used to handle the signal. Alternatively, if the implementation provides semaphores, they also can be used, either following *sigwait*( ) or from within a signal handling routine previously registered with *sigaction*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.4 (on page 478), Section 2.8.1 (on page 491), *pause*( ), *pthread_sigmask*( ), *sigaction*( ), *sigpending*( ), *sigsuspend*( ), *sigwaitinfo*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Issue 6

The RATIONALE section is added.

The **restrict** keyword is added to the *sigwait*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

sigwaitinfo — wait for queued signals (**REALTIME**)

SYNOPSIS

RTS

```
#include <signal.h>

int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
```

DESCRIPTION

Refer to *sigtimedwait*( ).

NAME

sin, sinf, sinl — sine function

SYNOPSIS

```
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the sine of their argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the sine of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error The *x* argument is ±Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

These functions may fail if:

MX Range Error The value of *x* is subnormal

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

Taking the Sine of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = sin(radians);
```

APPLICATION USAGE

These functions may lose accuracy when their argument is near a multiple of π or is far from 0.0.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asin( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The *sinf*( ) and *sinl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

sinf — sine function

SYNOPSIS

```
#include <math.h>

float sinf(float x);
```

DESCRIPTION

Refer to *sin*( ).

NAME

sinh, sinhf, sinhl — hyperbolic sine function

SYNOPSIS

```
#include <math.h>

double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the hyperbolic sine of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic sine of *x*.

If the result would cause an overflow, a range error shall occur and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as *x*) shall be returned as appropriate for the type of the function.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions shall fail if:

Range Error The result would cause an overflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

MX Range Error The value *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asinh( ), *cosh*( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tanh*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *sinhf*( ) and *sinhl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

sinl — sine function

SYNOPSIS

```
#include <math.h>

long double sinl(long double x);
```

DESCRIPTION

Refer to *sin*( ).

NAME

sleep — suspend execution for an interval of time

SYNOPSIS

```
#include <unistd.h>

unsigned sleep(unsigned seconds);
```

DESCRIPTION

The *sleep*( ) function shall cause the calling thread to be suspended from execution until either the number of realtime seconds specified by the argument *seconds* has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

If a SIGALRM signal is generated for the calling process during execution of *sleep*( ) and if the SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *sleep*( ) returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after *sleep*( ) returns or it is discarded.

If a SIGALRM signal is generated for the calling process during execution of *sleep*( ), except as a result of a prior call to *alarm*( ), and if the SIGALRM signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing *sleep*( ) to return.

If a signal-catching function interrupts *sleep*( ) and examines or changes either the time a SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts *sleep*( ) and calls *siglongjmp*( ) or *longjmp*( ) to restore an environment saved prior to the *sleep*( ) call, the action associated with the SIGALRM signal and the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored as part of the environment.

XSI Interactions between *sleep*( ) and any of *setitimer*( ), *ualarm*( ), or *usleep*( ) are unspecified.

RETURN VALUE

If *sleep*( ) returns because the requested time has elapsed, the value returned shall be 0. If *sleep*( ) returns due to delivery of a signal, the return value shall be the ‘‘unslept’’ amount (the requested time minus the time actually slept) in seconds.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

There are two general approaches to the implementation of the *sleep*( ) function. One is to use the *alarm*( ) function to schedule a SIGALRM signal and then suspend the process waiting for that signal. The other is to implement an independent facility. This volume of IEEE Std 1003.1-200x permits either approach.

In order to comply with the requirement that no primitive shall change a process attribute unless explicitly described by this volume of IEEE Std 1003.1-200x, an implementation using SIGALRM must carefully take into account any SIGALRM signal scheduled by previous *alarm*( ) calls, the action previously established for SIGALRM, and whether SIGALRM was blocked. If a SIGALRM has been scheduled before the *sleep*( ) would ordinarily complete, the *sleep*( ) must be shortened to that time and a SIGALRM generated (possibly simulated by direct invocation of the signal-catching function) before *sleep*( ) returns. If a SIGALRM has been scheduled after the *sleep*( ) would ordinarily complete, it must be rescheduled for the same time before *sleep*( ) returns. The action and blocking for SIGALRM must be saved and restored.

Historical implementations often implement the SIGALRM-based version using *alarm*( ) and *pause*( ). One such implementation is prone to infinite hangups, as described in *pause*( ). Another such implementation uses the C-language *setjmp*( ) and *longjmp*( ) functions to avoid that window. That implementation introduces a different problem: when the SIGALRM signal interrupts a signal-catching function installed by the user to catch a different signal, the *longjmp*( ) aborts that signal-catching function. An implementation based on *sigprocmask*( ), *alarm*( ), and *sigsuspend*( ) can avoid these problems.

Despite all reasonable care, there are several very subtle, but detectable and unavoidable, differences between the two types of implementations. These are the cases mentioned in this volume of IEEE Std 1003.1-200x where some other activity relating to SIGALRM takes place, and the results are stated to be unspecified. All of these cases are sufficiently unusual as not to be of concern to most applications.

See also the discussion of the term *realtime* in *alarm*( ).

Since *sleep*( ) can be implemented using *alarm*( ), the discussion about alarms occurring early under *alarm*( ) applies to *sleep*( ) as well. |

Application writers should note that the type of the argument *seconds* and the return value of *sleep*( ) is **unsigned**. That means that a Strictly Conforming POSIX System Interfaces Application cannot pass a value greater than the minimum guaranteed value for {UINT_MAX}, which the ISO C standard sets as 65 535, and any application passing a larger value is restricting its portability. A different type was considered, but historical implementations, including those with a 16-bit **int** type, consistently use either **unsigned** or **int**.

Scheduling delays may cause the process to return from the *sleep*( ) function significantly after the requested time. In such cases, the return value should be set to zero, since the formula (requested time minus the time actually spent) yields a negative number and *sleep*( ) returns an **unsigned**.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *getitimer*( ), *nanosleep*( ), *pause*( ), *sigaction*( ), *sigsetjmp*( ), *ualarm*( ), *usleep*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

NAME

snprintf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int snprintf(char *restrict s, size_t n,
    const char *restrict format, ...);
```

DESCRIPTION

Refer to *fprintf*( ).

NAME

socket — create an endpoint for communication

SYNOPSIS

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

DESCRIPTION

The *socket*( ) function shall create an unbound socket in a communications domain, and return a file descriptor that can be used in later function calls that operate on sockets.

The *socket*( ) function takes the following arguments:

domain	Specifies the communications domain in which a socket is to be created.
type	Specifies the type of socket to be created.
protocol	Specifies a particular protocol to be used with the socket. Specifying a *protocol* of 0 causes *socket*( ) to use an unspecified default protocol appropriate for the requested socket type.

The *domain* argument specifies the address family used in the communications domain. The address families supported by the system are implementation-defined.

Symbolic constants that can be used for the domain argument are defined in the **<sys/socket.h>** header.

The *type* argument specifies the socket type, which determines the semantics of communication over the socket. The following socket types are defined; implementations may specify additional socket types:

SOCK_STREAM	Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.
SOCK_DGRAM	Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.
SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-mode transmission path for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address family. If the *protocol* argument is zero, the default protocol for this address family and type shall be used. The protocols supported by the system are implementation-defined.

The process may need to have appropriate privileges to use the *socket*( ) function or to create some sockets.

RETURN VALUE

Upon successful completion, *socket*( ) shall return a non-negative integer, the socket file descriptor. Otherwise, a value of −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *socket*( ) function shall fail if:

[EAFNOSUPPORT]
The implementation does not support the specified address family.

[EMFILE]	No more file descriptors are available for this process.
[ENFILE]	No more file descriptors are available for the system.
[EPROTONOSUPPORT]	The protocol is not supported by the address family, or the protocol is not supported by the implementation.
[EPROTOTYPE]	The socket type is not supported by the protocol.

The *socket*( ) function may fail if:

[EACCES]	The process does not have appropriate privileges.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The application can determine whether an address family is supported by trying to create a socket with *domain* set to the protocol in question.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

accept( ), *bind*( ), *connect*( ), *getsockname*( ), *getsockopt*( ), *listen*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *setsockopt*( ), *shutdown*( ), *socketpair*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<netinet/in.h>**, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

socketpair — create a pair of connected sockets

SYNOPSIS

```
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol,
    int socket_vector[2]);
```

DESCRIPTION

The *socketpair*( ) function shall create an unbound pair of connected sockets in a specified *domain*, of a specified *type*, under the protocol optionally specified by the *protocol* argument. The two sockets shall be identical. The file descriptors used in referencing the created sockets shall be returned in *socket_vector*[0] and *socket_vector*[1].

The *socketpair*( ) function takes the following arguments:

domain	Specifies the communications domain in which the sockets are to be created.
type	Specifies the type of sockets to be created.
protocol	Specifies a particular protocol to be used with the sockets. Specifying a *protocol* of 0 causes *socketpair*( ) to use an unspecified default protocol appropriate for the requested socket type.
socket_vector	Specifies a 2-integer array to hold the file descriptors of the created socket pair.

The *type* argument specifies the socket type, which determines the semantics of communications over the socket. The following socket types are defined; implementations may specify additional socket types:

SOCK_STREAM	Provides sequenced, reliable, bidirectional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.
SOCK_DGRAM	Provides datagrams, which are connectionless-mode, unreliable messages of fixed maximum length.
SOCK_SEQPACKET	Provides sequenced, reliable, bidirectional, connection-mode transmission paths for records. A record can be sent using one or more output operations and received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the *protocol* argument is non-zero, it shall specify a protocol that is supported by the address family. If the *protocol* argument is zero, the default protocol for this address family and type shall be used. The protocols supported by the system are implementation-defined.

The process may need to have appropriate privileges to use the *socketpair*( ) function or to create some sockets.

RETURN VALUE

Upon successful completion, this function shall return 0; otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *socketpair*( ) function shall fail if:

[EAFNOSUPPORT]
The implementation does not support the specified address family.

[EMFILE]	No more file descriptors are available for this process.
[ENFILE]	No more file descriptors are available for the system.
[EOPNOTSUPP]	The specified protocol does not permit creation of socket pairs.
[EPROTONOSUPPORT]	The protocol is not supported by the address family, or the protocol is not supported by the implementation.
[EPROTOTYPE]	The socket type is not supported by the protocol.

The *socketpair*( ) function may fail if:

[EACCES]	The process does not have appropriate privileges.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENOMEM]	Insufficient memory was available to fulfill the request.

EXAMPLES

None.

APPLICATION USAGE

The documentation for specific address families specifies which protocols each address family supports. The documentation for specific protocols specifies which socket types each protocol supports.

The *socketpair*( ) function is used primarily with UNIX domain sockets and need not be supported for other domains.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

socket( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/socket.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the XNS, Issue 5.2 specification.

NAME

sprintf — print formatted output

SYNOPSIS

```
#include <stdio.h>

int sprintf(char *restrict s, const char *restrict format, ...);
```

DESCRIPTION

Refer to *fprintf*( ).

NAME

sqrt, sqrtf, sqrtl — square root function

SYNOPSIS

```
#include <math.h>

double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the square root of their argument x, $\sqrt{x}$.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the square root of x.

MX For finite values of $x < -0$, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

MX If x is NaN, a NaN shall be returned.

If x is ±0, or +Inf, x shall be returned.

If x is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error The finite value of x is < -0, or x is –Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

EXAMPLES

Taking the Square Root of 9.0

```
#include <math.h>
...
double x = 9.0;
double result;
...
result = sqrt(x);
```

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**, **<stdio.h>** |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *sqrtf*( ) and *sqrtl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

srand — pseudo-random number generator

SYNOPSIS

```
#include <stdlib.h>

void srand(unsigned seed);
```

DESCRIPTION

Refer to *rand*( ).

NAME

srand48 — seed uniformly distributed double-precision pseudo-random number generator

SYNOPSIS

XSI

```
#include <stdlib.h>

void srand48(long seedval);
```

DESCRIPTION

Refer to *drand48*( ).

NAME

srandom — seed pseudo-random number generator

SYNOPSIS

XSI

```
#include <stdlib.h>

void srandom(unsigned seed);
```

DESCRIPTION

Refer to *initstate*( ).

NAME

sscanf — convert formatted input

SYNOPSIS

```
#include <stdio.h>

int sscanf(const char *restrict s, const char *restrict format, ...);
```

DESCRIPTION

Refer to *fscanf*( ).

NAME

stat — get file status

SYNOPSIS

```
#include <sys/stat.h>

int stat(const char *restrict path, struct stat *restrict buf);
```

DESCRIPTION

The *stat*( ) function shall obtain information about the named file and write it to the area pointed to by the *buf* argument. The *path* argument points to a pathname naming a file. Read, write, or execute permission of the named file is not required. An implementation that provides additional or alternate file access control mechanisms may, under implementation-defined conditions, cause *stat*( ) to fail. In particular, the system may deny the existence of the file specified by *path*.

If the named file is a symbolic link, the *stat*( ) function shall continue pathname resolution using the contents of the symbolic link, and shall return information pertaining to the resulting file if the file exists.

The *buf* argument is a pointer to a **stat** structure, as defined in the **<sys/stat.h>** header, into which information is placed concerning the file.

The *stat*( ) function shall update any time-related fields (as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update), before writing into the **stat** structure.

The structure members *st_mode*, *st_ino*, *st_dev*, *st_uid*, *st_gid*, *st_atime*, *st_ctime*, and *st_mtime* shall have meaningful values for all file types defined in this volume of IEEE Std 1003.1-200x. The value of the member *st_nlink* shall be set to the number of links to the file.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *stat*( ) function shall fail if:

[EACCES]	Search permission is denied for a component of the path prefix.
[EIO]	An error occurred while reading from the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EOVERFLOW]	The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *buf*.

The *stat*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

[EOVERFLOW] A value to be stored would overflow one of the members of the **stat** structure.

EXAMPLES

Obtaining File Status Information

The following example shows how to obtain file status information for a file named **/home/cnd/mod1**. The structure variable *buffer* is defined for the **stat** structure.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

struct stat buffer;
int         status;
...
status = stat("/home/cnd/mod1", &buffer);
```

Getting Directory Information

The following example fragment gets status information for each entry in a directory. The call to the *stat*( ) function stores file information in the **stat** structure pointed to by *statbuf*. The lines that follow the *stat*( ) call format the fields in the **stat** structure for presentation to the user of the program.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <locale.h>
#include <langinfo.h>
#include <stdio.h>
#include <stdint.h>

struct dirent  *dp;
struct stat     statbuf;
struct passwd  *pwd;
struct group   *grp;
struct tm      *tm;
char            datestring[256];
...
/* Loop through directory entries */
while ((dp = readdir(dir)) != NULL) {

    /* Get entry's information. */
    if (stat(dp->d_name, &statbuf) == -1)
        continue;

    /* Print out type, permissions, and number of links. */
    printf("%10.10s", sperm (statbuf.st_mode));
    printf("%4d", statbuf.st_nlink);
```

```
        /* Print out owners name if it is found using getpwuid(). */
        if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
            printf(" %-8.8s", pwd->pw_name);
        else
            printf(" %-8d", statbuf.st_uid);

        /* Print out group name if it’s found using getgrgid(). */
        if ((grp = getgrgid(statbuf.st_gid)) != NULL)
            printf(" %-8.8s", grp->gr_name);
        else
            printf(" %-8d", statbuf.st_gid);

        /* Print size of file. */
        printf(" %9jd", (intmax_t)statbuf.st_size);

        tm = localtime(&statbuf.st_mtime);

        /* Get localized date string. */
        strftime(datestring, sizeof(datestring), nl_langinfo(D_T_FMT), tm);

        printf(" %s %s\n", datestring, dp->d_name);
    }
```

APPLICATION USAGE

None.

RATIONALE

The intent of the paragraph describing ‘‘additional or alternate file access control mechanisms’’ is to allow a secure implementation where a process with a label that does not dominate the file’s label cannot perform a *stat*( ) function. This is not related to read permission; a process with a label that dominates the file’s label does not need read permission. An implementation that supports write-up operations could fail *fstat*( ) function calls even though it has a valid file descriptor open for writing.

FUTURE DIRECTIONS

None.

SEE ALSO

fstat( ), *lstat*( ), *readlink*( ), *symlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification: |

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [EIO] mandatory error condition is added.

- The [ELOOP] mandatory error condition is added.
- The [EOVERFLOW] mandatory error condition is added. This change is to support large files.
- The [ENAMETOOLONG] and the second [EOVERFLOW] optional error conditions are added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Details are added regarding the treatment of symbolic links.
- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *stat*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

statvfs — get file system information

SYNOPSIS

XSI

```
#include <sys/statvfs.h>

int statvfs(const char *restrict path, struct statvfs *restrict buf);
```

DESCRIPTION

Refer to *fstatvfs*( ).

NAME

stderr, stdin, stdout — standard I/O streams

SYNOPSIS

```
#include <stdio.h>

extern FILE *stderr, *stdin, *stdout;
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type **FILE**. The *fopen*( ) function shall create certain descriptive data for a stream and return a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **<stdio.h>** header and associated with the standard open files.

At program start-up, three streams shall be predefined and need not be opened explicitly: *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

CX The following symbolic values in **<unistd.h>** define the file descriptors that shall be associated with the C-language *stdin*, *stdout*, and *stderr* when the application is started:

STDIN_FILENO	Standard input value, *stdin*. Its value is 0.
STDOUT_FILENO	Standard output value, *stdout*. Its value is 1.
STDERR_FILENO	Standard error value, *stderr*. Its value is 2.

The *stderr* stream is expected to be open for reading and writing.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fclose( ), *feof*( ), *ferror*( ), *fileno*( ), *fopen*( ), *fread*( ), *fseek*( ), *getc*( ), *gets*( ), *popen*( ), *printf*( ), *putc*( ), *puts*( ), *read*( ), *scanf*( ), *setbuf*( ), *setvbuf*( ), *tmpfile*( ), *ungetc*( ), *vprintf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1.

Issue 6

Extensions beyond the ISO C standard are now marked.

A note that *stderr* is expected to be open for reading and writing is added to the DESCRIPTION.

NAME

strcasecmp, strncasecmp — case-insensitive string comparisons

SYNOPSIS

XSI
```
#include <strings.h>

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

The *strcasecmp*( ) function shall compare, while ignoring differences in case, the string pointed to by *s1* to the string pointed to by *s2*. The *strncasecmp*( ) function shall compare, while ignoring differences in case, not more than *n* bytes from the string pointed to by *s1* to the string pointed to by *s2*.

In the POSIX locale, *strcasecmp*( ) and *strncasecmp*( ) shall behave as if the strings had been converted to lowercase and then a byte comparison performed. The results are unspecified in other locales. |

RETURN VALUE

Upon completion, *strcasecmp*( ) shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is, ignoring case, greater than, equal to, or less than the string pointed to by *s2*, respectively.

Upon successful completion, *strncasecmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is, ignoring case, greater than, equal to, or less than the possibly null-terminated array pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<strings.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

strcat — concatenate two strings

SYNOPSIS

```
#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcat*( ) function shall append a copy of the string pointed to by *s2* (including the terminating null byte) to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strcat*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strcat*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strchr — string scanning operation

SYNOPSIS

```
#include <string.h>

char *strchr(const char *s, int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *strchr*( ) function shall locate the first occurrence of *c* (converted to an **unsigned char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

RETURN VALUE

Upon completion, *strchr*( ) shall return a pointer to the byte, or a null pointer if the byte was not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strrchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strcmp — compare two strings

SYNOPSIS

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcmp*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon completion, *strcmp*( ) shall return an integer greater than, equal to, or less than 0, if the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

Checking a Password Entry

The following example compares the information read from standard input to the value of the name of the user entry. If the *strcmp*( ) function returns 0 (indicating a match), a further check will be made to see if the user entered the proper old password. The *crypt*( ) function shall encrypt the old password entered by the user, using the value of the encrypted password in the **passwd** structure as the salt. If this value matches the value of the encrypted **passwd** in the structure, the entered password *oldpasswd* is the correct user's password. Finally, the program encrypts the new password so that it can store the information in the **passwd** structure.

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) == 0) {
        strcpy(savepasswd, crypt(newpasswd, user));
        p->pw_passwd = savepasswd;
        valid_change = 1;
    }
    else {
```

```
                fprintf(stderr, "Old password is not valid\n");
            }
        }
        ...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strcoll — string comparison using collating information

SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcoll*( ) function shall compare the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

CX The *strcoll*( ) function shall not change the setting of *errno* if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strcoll*( ), then check *errno*. |

RETURN VALUE

Upon successful completion, *strcoll*( ) shall return an integer greater than, equal to, or less than 0, according to whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale. CX On error, *strcoll*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *strcoll*( ) function may fail if:

CX [EINVAL] The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

EXAMPLES

Comparing Nodes

The following example uses an application-defined function, *node_compare*( ), to compare two nodes based on an alphabetical ordering of the *string* field.

```
#include <string.h>
...
struct node { /* These are stored in the table. */
    char *string;
    int length;
};
...
int node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
        ((const struct node *)node2)->string);
}
...
```

APPLICATION USAGE

The *strxfrm*( ) and *strcmp*( ) functions should be used for sorting large lists.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcmp( ), *strxfrm*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate that *errno* does not be changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] optional error condition is added.

An example is added.

NAME

strcpy — copy a string

SYNOPSIS

```
#include <string.h>

char *strcpy(char *restrict s1, const char *restrict s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcpy*( ) function shall copy the string pointed to by *s2* (including the terminating null byte) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strcpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Initializing a String

The following example copies the string `"----------"` into the *permstring* variable.

```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "----------");
...
```

Storing a Key and Data

The following example allocates space for a key using *malloc*( ) then uses *strcpy*( ) to place the key there. Then it allocates space for data using *malloc*( ), and uses *strcpy*( ) to place data there. (The user-defined function *dbfree*( ) frees memory previously allocated to an array of type **struct element ***.)

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
...
/* Structure used to read data and store it. */
struct element {
    char *key;
    char *data;
};

struct element *tbl, *curtbl;
char *key, *data;
int count;
...
void dbfree(struct element *, int);
```

```
        ...
        if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
            perror("malloc"); dbfree(tbl, count); return NULL;
        }
        strcpy(curtbl->key, key);

        if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
            perror("malloc"); free(curtbl->key); dbfree(tbl, count); return NULL;
        }
        strcpy(curtbl->data, data);
        ...
```

APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

This issue is aligned with the ISO C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strncpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strcpy*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strcspn — get length of a complementary substring

SYNOPSIS

```
#include <string.h>

size_t strcspn(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strcspn*( ) function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes *not* from the string pointed to by *s2*. |

RETURN VALUE

The *strcspn*( ) function shall return the length of the computed segment of the string pointed to by *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The RETURN VALUE section is updated to indicated that *strcspn*( ) returns the length of *s1*, and not *s1* itself as was previously stated.

Issue 6

The Open Group Corrigendum U030/1 is applied. The text of the RETURN VALUE section is updated to indicate that the computed segment length is returned, not the *s1* length.

NAME

strdup — duplicate a string

SYNOPSIS

XSI

```
#include <string.h>

char *strdup(const char *s1);
```

DESCRIPTION

The *strdup*( ) function shall return a pointer to a new string, which is a duplicate of the string pointed to by *s1*. The returned pointer can be passed to *free*( ). A null pointer is returned if the new string cannot be created.

RETURN VALUE

The *strdup*( ) function shall return a pointer to a new string on success. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *strdup*( ) function may fail if:

[ENOMEM] Storage space available is insufficient.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

free( ), *malloc*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

strerror, strerror_r — get error message string

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

DESCRIPTION

For *strerror*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strerror*( ) function shall map the error number in *errnum* to a locale-dependent error message string and shall return a pointer to it. Typically, the values for *errnum* come from *errno*, but *strerror*( ) shall map any value of type **int** to a message.

The string pointed to shall not be modified by the application, but may be overwritten by a subsequent call to *strerror*( ) or *perror*( ).

The contents of the error message strings returned by *strerror*( ) should be determined by the setting of the *LC_MESSAGES* category in the current locale.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *strerror*( ).

The *strerror*( ) function shall not change the setting of *errno* if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strerror*( ), then check *errno*. |

The *strerror*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The *strerror_r*( ) function shall map the error number in *errnum* to a locale-dependent error message string and shall return the string in the buffer pointed to by *strerrbuf*, with length *buflen*.

RETURN VALUE

Upon successful completion, *strerror*( ) shall return a pointer to the generated message string. On error *errno* may be set, but no return value is reserved to indicate an error.

Upon successful completion, *strerror_r*( ) shall return 0. Otherwise, an error number shall be returned to indicate the error.

ERRORS

These functions may fail if:

[EINVAL]	The value of *errnum* is not a valid error number.

The *strerror_r*( ) function may fail if:

[ERANGE]	Insufficient storage was supplied via *strerrbuf* and *buflen* to contain the generated message string.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

perror( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

A note indicating that this function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE section, the fact that *errno* may be set is added.
- The [EINVAL] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *strerror_r*( ) function is added in response to IEEE PASC Interpretation 1003.1c #39.

The *strerror_r*( ) function is marked as part of the Thread-Safe Functions option.

NAME

strfmon — convert monetary value to a string

SYNOPSIS

XSI
```
#include <monetary.h>

ssize_t strfmon(char *restrict s, size_t maxsize,
    const char *restrict format, ...);
```

DESCRIPTION

The *strfmon*( ) function shall place characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string, beginning and ending in its initial state, if any, that contains two types of objects: *plain characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

The application shall ensure that a conversion specification consists of the following sequence:

- A '%' character
- Optional flags
- Optional field width
- Optional left precision
- Optional right precision
- A required conversion specifier character that determines the conversion to be performed

Flags

One or more of the following optional flags can be specified to control the conversion:

=*f*	An '=' followed by a single character *f* which is used as the numeric fill character. In order to work with precision or width counts, the fill character shall be a single byte character; if not, the behavior is undefined. The default numeric fill character is the <space>. This flag does not affect field width filling which always uses the <space>. This flag is ignored unless a left precision (see below) is specified.
^	Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.
+ or (	Specify the style of representing positive and negative currency amounts. Only one of '+' or '(' may be specified. If '+' is specified, the locale's equivalent of '+' and '−' are used (for example, in the U.S., the empty string if positive and '−' if negative). If '(' is specified, negative amounts are enclosed within parentheses. If neither flag is specified, the '+' style is used.
!	Suppress the currency symbol from the output conversion.
−	Specify the alignment. If this flag is present the result of the conversion is left-justified (padded to the right) rather than right-justified. This flag shall be ignored unless a field width (see below) is specified.

Field Width

w A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag `'−'` is specified). The default is 0.

Left Precision

#*n* A `'#'` followed by a decimal digit string *n* specifying a maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the *strfmon*( ) function aligned in the same columns. It can also be used to fill unused positions with a special character as in `"$***123.45"`. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the =*f* flag above).

If grouping has not been suppressed with the `'^'` flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with <space>s to make their positive and negative formats an equal length.

Right Precision

.*p* A period followed by a decimal digit string *p* specifying the number of digits after the radix character. If the value of the right precision *p* is 0, no radix character appears. If a right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Specifier Characters

The conversion specifier characters and their meanings are:

`i` The **double** argument is formatted according to the locale's international currency format (for example, in the U.S.: USD 1,234.56). If the argument is ±Inf or NaN, the result of the conversion is unspecified.

`n` The **double** argument is formatted according to the locale's national currency format (for example, in the U.S.: $1,234.56). If the argument is ±Inf or NaN, the result of the conversion is unspecified.

`%` Convert to a `'%'`; no argument is converted. The entire conversion specification shall be %%.

Locale Information

The *LC_MONETARY* category of the program's locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the *LC_NUMERIC* category), the grouping separator, the currency symbols, and formats. The international currency symbol should be conformant with the ISO 4217: 1995 standard.

If the value of *maxsize* is greater than {SSIZE_MAX}, the result is implementation-defined.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strfmon*( ) shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, –1 shall be returned, the contents of the array are unspecified, and *errno* shall be set to indicate the error.

ERRORS

The *strfmon*( ) function shall fail if:

[E2BIG]	Conversion stopped due to lack of space in the buffer.

EXAMPLES

Given a locale for the U.S. and the values 123.45, –123.45, and 3456.781:

Conversion Specification	Output	Comments

`%n`	`$123.45`
	`-$123.45`
	`$3,456.78`
		Default formatting.

`%11n`	`$123.45`
	`-$123.45`
	`$3,456.78`
		Right align within an 11 character field.

`%#5n`	`$    123.45`
	`-$    123.45`
	`$ 3,456.78`
		Aligned columns for values up to 99,999.

`%=*#5n`	`$***123.45`
	`-$***123.45`
	`$*3,456.78`
		Specify a fill character.

`%=0#5n`	`$000123.45`
	`-$000123.45`
	`$03,456.78`
		Fill characters do not use grouping even if the fill character is a digit.

`%^#5n`	`$  123.45`
	`-$  123.45`
	`$ 3456.78`
		Disable the grouping separator.

`%^#5.0n`	`$  123`
	`-$  123`
	`$ 3457`
		Round off to whole units.

`%^#5.4n`	`$  123.4500`
	`-$  123.4500`
	`$ 3456.7810`
		Increase the precision.

`%(#5n`	`$    123.45`
	`($    123.45)`
	`$ 3,456.78`
		Use an alternative pos/neg style.

`%(!#5n`	`123.45`
	`(    123.45)`
	`3,456.78`
		Disable the currency symbol.

`%-14#5.4n`	`$  123.4500`
	`-$  123.4500`
	`$3,456.7810`
		Left-justify the output.

`%14#5.4n`	`$  123.4500`
	`-$  123.4500`
	`$3,456.7810`
		Corresponding right-justified output.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

Lowercase conversion characters are reserved for future standards use and uppercase for implementation-defined use.

SEE ALSO

localeconv( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<monetary.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE.

The [ENOSYS] error is removed.

A sentence is added to the DESCRIPTION warning about values of *maxsize* that are greater than {SSIZE_MAX}.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *strfmon*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

strftime — convert date and time to a string

SYNOPSIS

```
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize,
    const char *restrict format, const struct tm *restrict timeptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strftime*( ) function shall place bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The format is a character string, beginning and ending in its initial shift state, if any. The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a `'%'` character, possibly followed by an `E` or `O` modifier, and a terminating conversion specifier character that determines the conversion specification's behavior. All ordinary characters (including the terminating null byte) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than *maxsize* bytes are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined using the *LC_TIME* category of the current locale and by the values of zero or more members of the broken-down time structure pointed to by *timeptr*, as specified in brackets in the description. If any of the specified values are outside the normal range, the characters stored are unspecified.

CX Local timezone information is used as though *strftime*( ) called *tzset*( ).

The following conversion specifications are supported:

`%a`	Replaced by the locale's abbreviated weekday name. [*tm_wday*]
`%A`	Replaced by the locale's full weekday name. [*tm_wday*]
`%b`	Replaced by the locale's abbreviated month name. [*tm_mon*]
`%B`	Replaced by the locale's full month name. [*tm_mon*]
`%c`	Replaced by the locale's appropriate date and time representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
`%C`	Replaced by the year divided by 100 and truncated to an integer, as a decimal number [00,99]. [*tm_year*]
`%d`	Replaced by the day of the month as a decimal number [01,31]. [*tm_mday*]
`%D`	Equivalent to `%m/%d/%y`. [*tm_mon*, *tm_mday*, *tm_year*]
`%e`	Replaced by the day of the month as a decimal number [1,31]; a single digit is preceded by a space. [*tm_mday*]
`%F`	Equivalent to `%Y-%m-%d` (the ISO 8601:2000 standard date format). [*tm_year*, *tm_mon*, *tm_mday*]
`%g`	Replaced by the last 2 digits of the week-based year (see below) as a decimal number [00,99]. [*tm_year*, *tm_wday*, *tm_yday*]
`%G`	Replaced by the week-based year (see below) as a decimal number (for example, 1977). [*tm_year*, *tm_wday*, *tm_yday*]

	`%h`	Equivalent to `%b`. [*tm_mon*]
	`%H`	Replaced by the hour (24-hour clock) as a decimal number [00,23]. [*tm_hour*]
	`%I`	Replaced by the hour (12-hour clock) as a decimal number [01,12]. [*tm_hour*]
	`%j`	Replaced by the day of the year as a decimal number [001,366]. [*tm_yday*]
	`%m`	Replaced by the month as a decimal number [01,12]. [*tm_mon*]
	`%M`	Replaced by the minute as a decimal number [00,59]. [*tm_min*]
	`%n`	Replaced by a <newline>.
	`%p`	Replaced by the locale's equivalent of either a.m. or p.m. [*tm_hour*]
CX	`%r`	Replaced by the time in a.m. and p.m. notation; in the POSIX locale this shall be equivalent to `%I:%M:%S %p`. [*tm_hour*, *tm_min*, *tm_sec*]
	`%R`	Replaced by the time in 24 hour notation (`%H:%M`). [*tm_hour*, *tm_min*]
	`%S`	Replaced by the second as a decimal number [00,60]. [*tm_sec*]
	`%t`	Replaced by a <tab>.
	`%T`	Replaced by the time (`%H:%M:%S`). [*tm_hour*, *tm_min*, *tm_sec*]
	`%u`	Replaced by the weekday as a decimal number [1,7], with 1 representing Monday. [*tm_wday*]
	`%U`	Replaced by the week number of the year as a decimal number [00,53]. The frist Sunday of January is the first day of week 1; days in the new year before this are in week 0. [*tm_year*, *tm_wday*, *tm_yday*]
	`%V`	Replaced by the week number of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1. [*tm_year*, *tm_wday*, *tm_yday*]
	`%w`	Replaced by the weekday as a decimal number [0,6], with 0 representing Sunday. [*tm_wday*]
	`%W`	Replaced by the week number of the year as a decimal number [00,53]. The first Monday of January is the first day of week 1; days in the new year before this are in week 0. [*tm_year*, *tm_wday*, *tm_yday*]
	`%x`	Replaced by the locale's appropriate date representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
	`%X`	Replaced by the locale's appropriate time representation. (See the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**.)
	`%y`	Replaced by the last two digits of the year as a decimal number [00,99]. [*tm_year*]
	`%Y`	Replaced by the year as a decimal number (for example, 1997). [*tm_year*]
CX	`%z`	Replaced by the offset from UTC in the ISO 8601:2000 standard format (`+hhmm` or `−hhmm`), or by no characters if no timezone is determinable. For example, `"−0430"` means 4 hours 30 minutes behind UTC (west of Greenwich). If *tm_isdst* is zero, the standard time offset is used. If *tm_isdst* is greater than zero, the daylight savings time offset is used. If *tm_isdst* is negative, no characters are returned. [*tm_isdst*]

`%Z`	Replaced by the timezone name or abbreviation, or by no bytes if no timezone information exists. [*tm_isdst*]
`%%`	Replaced by `%`.

If a conversion specification does not correspond to any of the above, the behavior is undefined.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the `E` or `O` modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist for the current locale, (see ERA in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5, LC_TIME) the behavior shall be as if the unmodified conversion specification were used.

`%Ec`	Replaced by the locale's alternative appropriate date and time representation.
`%EC`	Replaced by the name of the base year (period) in the locale's alternative representation.
`%Ex`	Replaced by the locale's alternative date representation.
`%EX`	Replaced by the locale's alternative time representation.
`%Ey`	Replaced by the offset from `%EC` (year only) in the locale's alternative representation.
`%EY`	Replaced by the full alternative year representation.
`%Od`	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero; otherwise, with leading spaces.
`%Oe`	Replaced by the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
`%OH`	Replaced by the hour (24-hour clock) using the locale's alternative numeric symbols.
`%OI`	Replaced by the hour (12-hour clock) using the locale's alternative numeric symbols.
`%Om`	Replaced by the month using the locale's alternative numeric symbols.
`%OM`	Replaced by the minutes using the locale's alternative numeric symbols.
`%OS`	Replaced by the seconds using the locale's alternative numeric symbols.
`%Ou`	Replaced by the weekday as a number in the locale's alternative representation (Monday=1).
`%OU`	Replaced by the week number of the year (Sunday as the first day of the week, rules corresponding to `%U`) using the locale's alternative numeric symbols.
`%OV`	Replaced by the week number of the year (Monday as the first day of the week, rules corresponding to `%V`) using the locale's alternative numeric symbols.
`%Ow`	Replaced by the number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
`%OW`	Replaced by the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
`%Oy`	Replaced by the year (offset from `%C`) using the locale's alternative numeric symbols.

`%g`, `%G`, and `%V` give values according to the ISO 8601:2000 standard week-based year. In this system, weeks begin on a Monday and week 1 of the year is the week that includes January 4th, which is also the week that includes the first Thursday of the year, and is also the first week that contains at least four days in the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year; thus, for Saturday 2nd January 1999, `%G` is replaced by 1998 and `%V` is replaced by 53. If December 29th, 30th, or 31st is a Monday, it and any following days are part of week 1 of the following year. Thus, for Tuesday 30th December 1997, `%G` is replaced by 1998 and `%V` is replaced by 01.

If a conversion specifier is not one of the above, the behavior is undefined. |

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, *strftime*( ) shall return the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, 0 shall be returned and the contents of the array are unspecified. |

ERRORS

No errors are defined.

EXAMPLES

Getting a Localized Date String

The following example first sets the locale to the user's default. The locale information will be used in the *nl_langinfo*( ) and *strftime*( ) functions. The *nl_langinfo*( ) function returns the localized date string which specifies how the date is laid out. The *strftime*( ) function takes this information and, using the **tm** structure for values, places the date and time information into *datestring*.

```
#include <time.h>
#include <locale.h>
#include <langinfo.h>
...
struct tm *tm;
char datestring[256];
...
setlocale (LC_ALL, "");
...
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT), tm);
...
```

APPLICATION USAGE

The range of values for `%S` is [00,60] rather than [00,59] to allow for the occasional leap second.

Some of the conversion specifications are duplicates of others. They are included for compatibility with *nl_cxtime*( ) and *nl_ascxtime*( ), which were published in Issue 2. |

Applications should use `%Y` (4-digit years) in preference to `%y` (2-digit years).

In the C locale, the `E` and `O` modifiers are ignored and the replacement strings for the following specifiers are:

`%a`	The first three characters of `%A`.
`%A`	One of Sunday, Monday, . . ., Saturday.
`%b`	The first three characters of `%B`.
`%B`	One of January, February, . . ., December.
`%c`	Equivalent to `%a %b %e %T %Y`.

`%p`	One of AM or PM.
`%r`	Equivalent to `%I:%M:%S %p`.
`%x`	Equivalent to `%m/%d/%y`.
`%X`	Equivalent to `%T`.
`%Z`	Implementation-defined.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *getdate*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strptime*( ), *time*( ), *tzset*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 5

The description of `%OV` is changed to be consistent with `%V` and defines Monday as the first day of the week.

The description of `%Oy` is clarified.

Issue 6

Extensions beyond the ISO C standard are now marked.

The Open Group Corrigendum U033/8 is applied. The `%V` conversion specifier is changed from ''Otherwise, it is week 53 of the previous year, and the next week is week 1'' to ''Otherwise, it is the last week of the previous year, and the next week is week 1''.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The `%C`, `%D`, `%e`, `%h`, `%n`, `%r`, `%R`, `%t`, and `%T` conversion specifiers are added.
- The modified conversion specifiers are added for consistency with the ISO POSIX-2 standard *date* utility.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *strftime*( ) prototype is updated.
- The DESCRIPTION is extensively revised.
- The `%z` conversion specifier is added.

A new example is added.

NAME

strlen — get string length

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strlen*( ) function shall compute the number of bytes in the string to which *s* points, not including the terminating null byte.

RETURN VALUE

The *strlen*( ) function shall return the length of *s*; no return value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

Getting String Lengths

The following example sets the maximum length of *key* and *data* by using *strlen*( ) to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The RETURN VALUE section is updated to indicate that *strlen*( ) returns the length of *s*, and not *s* itself as was previously stated.

NAME

strncasecmp — case-insensitive string comparison

SYNOPSIS

XSI

```
#include <strings.h>

int strncasecmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

Refer to *strcasecmp*( ).

NAME

strncat — concatenate a string with part of another

SYNOPSIS

```
#include <string.h>

char *strncat(char *restrict s1, const char *restrict s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strncat*( ) function shall append not more than *n* bytes (a null byte and bytes that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial byte of *s2* overwrites the null byte at the end of *s1*. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *strncat*( ) function shall return *s1*; no return value shall be reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strncat*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

strncmp — compare part of two strings

SYNOPSIS

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strncmp*( ) function shall compare not more than *n* bytes (bytes that follow a null byte are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE

Upon successful completion, *strncmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *s2* respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

strncpy — copy part of a string

SYNOPSIS

```
#include <string.h>

char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strncpy*( ) function shall copy not more than *n* bytes (bytes that follow a null byte are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by *s2* is a string that is shorter than *n* bytes, null bytes shall be appended to the copy in the array pointed to by *s1*, until *n* bytes in all are written.

RETURN VALUE

The *strncpy*( ) function shall return *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Character movement is performed differently in different implementations. Thus, overlapping moves may yield surprises.

If there is no null byte in the first *n* bytes of the array pointed to by *s2*, the result is not null-terminated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The *strncpy*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

strpbrk — scan string for byte

SYNOPSIS

```
#include <string.h>

char *strpbrk(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strpbrk*( ) function shall locate the first occurrence in the string pointed to by *s1* of any byte from the string pointed to by *s2*.

RETURN VALUE

Upon successful completion, *strpbrk*( ) shall return a pointer to the byte or a null pointer if no byte from *s2* occurs in *s1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), *strrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

strptime — date and time conversion

SYNOPSIS

XSI
```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
    struct tm *restrict tm);
```

DESCRIPTION

The *strptime*( ) function shall convert the character string pointed to by *buf* to values which are stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

The *format* is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by *isspace*( )); an ordinary character (neither `'%'` nor a white-space character); or a conversion specification. Each conversion specification is composed of a `'%'` character followed by a conversion character which specifies the replacement required. The application shall ensure that there is white-space or other non-alphanumeric characters between any two conversion specifications. The following conversion specifications are supported:

`%a`	The day of the week, using the locale's weekday names; either the abbreviated or full name may be specified.
`%A`	Equivalent to `%a`.
`%b`	The month, using the locale's month names; either the abbreviated or full name may be specified.
`%B`	Equivalent to `%b`.
`%c`	Replaced by the locale's appropriate date and time representation.
`%C`	The century number [0,99]; leading zeros are permitted but not required.
`%d`	The day of the month [1,31]; leading zeros are permitted but not required.
`%D`	The date as `%m/%d/%y`.
`%e`	Equivalent to `%d`.
`%h`	Equivalent to `%b`.
`%H`	The hour (24-hour clock) [0,23]; leading zeros are permitted but not required.
`%I`	The hour (12-hour clock) [1,12]; leading zeros are permitted but not required.
`%j`	The day number of the year [1,366]; leading zeros are permitted but not required.
`%m`	The month number [1,12]; leading zeros are permitted but not required.
`%M`	The minute [0,59]; leading zeros are permitted but not required.
`%n`	Any white space.
`%p`	The locale's equivalent of a.m or p.m.
`%r`	12-hour clock time using the AM / PM notation if **t_fmt_ampm** is not an empty string in the LC_TIME portion of the current locale; in the POSIX locale, this shall be equivalent to `%I:%M:%S %p`.
`%R`	The time as `%H:%M`.

`%S`	The seconds [0,60]; leading zeros are permitted but not required.
`%t`	Any white space.
`%T`	The time as `%H:%M:%S`.
`%U`	The week number of the year (Sunday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.
`%w`	The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required.
`%W`	The week number of the year (Monday as the first day of the week) as a decimal number [00,53]; leading zeros are permitted but not required.
`%x`	The date, using the locale's date format.
`%X`	The time, using the locale's time format.
`%y`	The year within century. When a century is not otherwise specified, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive; leading zeros shall be permitted but shall not be required. **Note:** It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)
`%Y`	The year, including the century (for example, 1988).
`%%`	Replaced by `%`.

Modified Conversion Specifiers

Some conversion specifiers can be modified by the `E` and `O` modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specifier. If the alternative format or specification does not exist in the current locale, the behavior shall be as if the unmodified conversion specification were used.

`%Ec`	The locale's alternative appropriate date and time representation.
`%EC`	The name of the base year (period) in the locale's alternative representation.
`%Ex`	The locale's alternative date representation.
`%EX`	The locale's alternative time representation.
`%Ey`	The offset from `%EC` (year only) in the locale's alternative representation.
`%EY`	The full alternative year representation.
`%Od`	The day of the month using the locale's alternative numeric symbols; leading zeros are permitted but not required.
`%Oe`	Equivalent to `%Od`.
`%OH`	The hour (24-hour clock) using the locale's alternative numeric symbols.
`%OI`	The hour (12-hour clock) using the locale's alternative numeric symbols.
`%Om`	The month using the locale's alternative numeric symbols.
`%OM`	The minutes using the locale's alternative numeric symbols.

`%OS`	The seconds using the locale's alternative numeric symbols.
`%OU`	The week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
`%Ow`	The number of the weekday (Sunday=0) using the locale's alternative numeric symbols.
`%OW`	The week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
`%Oy`	The year (offset from `%C`) using the locale's alternative numeric symbols.

A conversion specification composed of white-space characters is executed by scanning input up to the first character that is not white-space (which remains unscanned), or until no more characters can be scanned.

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of conversion specifications composed of `%n`, `%t`, white-space characters, or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, except the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate **tm** structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf* such as month or weekday names. If no match is found, *strptime*( ) fails and no more characters are scanned.

RETURN VALUE

Upon successful completion, *strptime*( ) shall return a pointer to the character following the last character parsed. Otherwise, a null pointer shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Several ''equivalent to'' formats and the special processing of white-space characters are provided in order to ease the use of identical *format* strings for *strftime*( ) and *strptime*( ). |

Applications should use `%Y` (4-digit years) in preference to `%y` (2-digit years).

It is unspecified whether multiple calls to *strptime*( ) using the same **tm** structure will update the current contents of the structure or overwrite all contents of the structure. Conforming applications should make a single call to *strptime*( ) with a format and all data needed to completely specify the date and time being converted. |

RATIONALE

None.

FUTURE DIRECTIONS

The *strptime*( ) function is expected to be mandatory in the next version of this volume of IEEE Std 1003.1-200x.

SEE ALSO

scanf( ), *strftime*( ), *time*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE.

The [ENOSYS] error is removed.

The exact meaning of the `%y` and `%Oy` specifiers are clarified in the DESCRIPTION.

Issue 6

The Open Group Corrigendum U033/5 is applied. The `%r` specifier description is reworded.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *strptime*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Corrigendum U047/2 is applied.

The DESCRIPTION is updated to use the terms ‘‘conversion specifier’’ and ‘‘conversion specification’’ for consistency with *strftime*( ).

NAME

strrchr — string scanning operation

SYNOPSIS

```
#include <string.h>

char *strrchr(const char *s, int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *strrchr*( ) function shall locate the last occurrence of *c* (converted to an **unsigned char**) in the string pointed to by *s*. The terminating null byte is considered to be part of the string.

RETURN VALUE

Upon successful completion, *strrchr*( ) shall return a pointer to the byte or a null pointer if *c* does not occur in the string.

ERRORS

No errors are defined.

EXAMPLES

Finding the Base Name of a File

The following example uses *strrchr*( ) to get a pointer to the base name of a file. The *strrchr*( ) function searches backwards through the name of the file to find the last '/' character in *name*. This pointer (plus one) will point to the base name of the file.

```
#include <string.h>
...
const char *name;
char *basename;
...
basename = strrchr(name, '/') + 1;
...
```

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

strspn — get length of a substring

SYNOPSIS

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strspn*( ) function shall compute the length (in bytes) of the maximum initial segment of the string pointed to by *s1* which consists entirely of bytes from the string pointed to by *s2*. |

RETURN VALUE

The *strspn*( ) function shall return the length of *s1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The RETURN VALUE section is updated to indicate that *strspn*( ) returns the length of *s*, and not *s* itself as was previously stated.

NAME

strstr — find a substring

SYNOPSIS

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strstr*( ) function shall locate the first occurrence in the string pointed to by *s1* of the sequence of bytes (excluding the terminating null byte) in the string pointed to by *s2*.

RETURN VALUE

Upon successful completion, *strstr*( ) shall return a pointer to the located string or a null pointer if the string is not found.

If *s2* points to a string with zero length, the function shall return *s1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ANSI C standard.

NAME

strtod, strtof, strtold — convert string to a double-precision number

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr);
float strtof(const char *restrict nptr, char **restrict endptr);
long double strtold(const char *restrict nptr, char **restrict endptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequence*$_{opt}$), ignoring case in the NAN part, where:

```
n-char-sequence:
    digit
    nondigit
    n-char-sequence digit
    n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) shall be interpreted as a floating constant of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence shall be interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequence*$_{opt}$) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the value resulting from the conversion is correctly rounded.

CX The radix character is defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`).

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *strtod*( ) function shall not change the setting of *errno* if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *strtod*( ), *strtof*( ), or *strtold*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned, and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, HUGE_VAL, HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [ERANGE] The value to be returned would cause overflow or underflow.

These functions may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, adn the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in **<float.h>**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than DECIMAL_DIG significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the values of L, D, and U satisfy $L <= D <= U$. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding L and U according to the current rounding direction, with the extra stipulation that the error with respect to D should have a correct sign for the current rounding direction.

The changes to *strtod*( ) introduced by the ISO / IEC 9899: 1999 standard can alter the behavior of well-formed applications complying with the ISO / IEC 9899: 1990 standard and thus earlier versions of IEEE Std 1003.1-200x. One such example would be:

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
    long l;

    d = strtod(s, &endp);
    if (s != endp && *endp == '\0')
        printf("It's a float with value %g\n", d);
    else
    {
        l = strtol(s, &endp, 0);
        if (s != endp && *endp == '\0')
            printf("It's an integer with value %ld\n", 1);
        else
            return 1;
    }
    return 0;
}
```

If the function is called with:

```
what_kind_of_number ("0x10")
```

an ISO / IEC 9899: 1990 standard-compliant library will result in the function printing:

```
It's an integer with value 16
```

With the ISO / IEC 9899: 1999 standard, the result is:

```
It's a float with value 16
```

The change in behavior is due to the inclusion of floating-point numbers in hexadecimal notation without requiring that either a decimal point or the binary exponent be present.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isspace( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<float.h>**, **<stdlib.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *strtod*( ) function is updated.
- The *strtof*( ) and *strtold*( ) functions are added.
- The DESCRIPTION is extensively revised.

ISO / IEC 9899: 1999 standard, Technical Corrigendum No. 1 is incorporated. |

NAME

strtof — convert string to a double-precision number

SYNOPSIS

```
#include <stdlib.h>

float strtof(const char *restrict nptr, char **restrict endptr);
```

DESCRIPTION

Refer to *strtod*( ).

NAME

strtoimax, strtoumax — convert string to integer type

SYNOPSIS

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr, char **restrict endptr,
    int base);
uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
    int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall be equivalent to the *strtol*( ), *strtoll*( ), *strtoul*( ), and *strtoull*( ) functions, except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

RETURN VALUE

These functions shall return the converted value, if any.

If no conversion could be performed, zero shall be returned.

If the correct value is outside the range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign of the value, if any), and *errno* shall be set to [ERANGE].

ERRORS

These functions shall fail if:

[ERANGE]	The value to be returned is not representable.

These functions may fail if:

[EINVAL]	The value of *base* is not supported.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strtol( ), *strtoul*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

strtok, strtok_r — split string into tokens

SYNOPSIS

```
#include <string.h>

char *strtok(char *restrict s1, const char *restrict s2);
char *strtok_r(char *restrict s, const char *restrict sep,
    char **restrict lasts);
```

TSF

DESCRIPTION

CX For *strtok*( ): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

A sequence of calls to *strtok*( ) breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a byte from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* may be different from call to call.

The first call in the sequence searches the string pointed to by *s1* for the first byte that is *not* contained in the current separator string pointed to by *s2*. If no such byte is found, then there are no tokens in the string pointed to by *s1* and *strtok*( ) shall return a null pointer. If such a byte is found, it is the start of the first token.

The *strtok*( ) function then searches from there for a byte that *is* contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The *strtok*( ) function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *strtok*( ).

CX The *strtok*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *strtok_r*( ) function considers the null-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. The argument *lasts* points to a user-provided pointer which points to stored information necessary for *strtok_r*( ) to continue scanning the same string.

In the first call to *strtok_r*( ), *s* points to a null-terminated string, *sep* to a null-terminated string of separator characters, and the value pointed to by *lasts* is ignored. The *strtok_r*( ) function shall return a pointer to the first character of the first token, write a null character into *s* immediately following the returned token, and update the pointer to which *lasts* points.

In subsequent calls, *s* is a NULL pointer and *lasts* shall be unchanged from the previous call so that subsequent calls shall move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no token remains in *s*, a NULL pointer shall be returned.

RETURN VALUE

Upon successful completion, *strtok*( ) shall return a pointer to the first byte of a token. Otherwise, if there is no token, *strtok*( ) shall return a null pointer.

TSF The *strtok_r*( ) function shall return a pointer to the token found, or a NULL pointer when no token is found.

ERRORS

No errors are defined.

EXAMPLES

Searching for Word Separators

The following example searches for tokens separated by space characters.

```
#include <string.h>
...
char *token;
char *line = "LINE TO BE SEPARATED";
char *search = " ";

/* Token will point to "LINE". */
token = strtok(line, search);

/* Token will point to "TO". */
token = strtok(NULL, search);
```

Breaking a Line

The following example uses *strtok*( ) to break a line into two character strings separated by any combination of <space>s, <tab>s, or <newline>s.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char line[LINE_MAX];
char *key, *data;
...
key = strtok(line, " \n");
data = strtok(NULL, " \n");
...
```

APPLICATION USAGE

The *strtok_r*( ) function is thread-safe and stores its state in a user-supplied buffer instead of possibly using a static data area that may be overwritten by an unrelated call from another thread.

RATIONALE

The *strtok*( ) function searches for a separator string within a larger string. It returns a pointer to the last substring between separator strings. This function uses static storage to keep track of the current string position between calls. The new function, *strtok_r*( ), takes an additional argument, *lasts*, to keep track of the current position in the string.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The *strtok_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *strtok*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

Extensions beyond the ISO C standard are now marked.

The *strtok_r*( ) function is marked as part of the Thread-Safe Functions option.

In the DESCRIPTION, the note about reentrancy is expanded to cover thread-safety.

The APPLICATION USAGE section is updated to include a note on the thread-safe function and its avoidance of possibly using a static data area.

The **restrict** keyword is added to the *strtok*( ) and *strtok_r*( ) prototypes for alignment with the ISO/IEC 9899:1999 standard.

NAME

strtol, strtoll — convert string to a long integer

SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *restrict str, char **restrict endptr, int base);
long long strtoll(const char *restrict str, char **restrict endptr,
    int base)
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by *str* to a type **long** and **long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

Then they shall attempt to convert the subject sequence to an integer, and return the result. |

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a `'+'` or `'−'` sign. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall | contain no characters if the input string is empty or consists entirely of white-space characters, | or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of | characters starting with the first digit shall be interpreted as an integer constant. If the subject | sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the | base for conversion, ascribing to each letter its value as given above. If the subject sequence | begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to | the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null | pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *strtol*( ) function shall not change the setting of *errno* if successful.

Since 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *strtol*( ) or *strtoll*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned CX and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX}, {LLONG_MIN} or {LLONG_MAX} shall be returned (according to the sign of the value), and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [EINVAL] The value of *base* is not supported.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *scanf*( ), *strtod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *strtol*( ) prototype is updated.
- The *strtoll*( ) function is added.

NAME

strtold — convert string to a double-precision number

SYNOPSIS

```
#include <stdlib.h>

long double strtold(const char *restrict nptr, char **restrict endptr);
```

DESCRIPTION

Refer to *strtod*( ).

NAME

strtoll — convert string to a long integer

SYNOPSIS

```
#include <stdlib.h>

long long strtoll(const char *restrict str, char **restrict endptr,
    int base);
```

DESCRIPTION

Refer to *strtol*( ).

NAME

strtoul, strtoull — convert string to an unsigned long

SYNOPSIS

```
#include <stdlib.h>

unsigned long strtoul(const char *restrict str,
    char **restrict endptr, int base);
unsigned long long strtoull(const char *restrict str,
    char **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the string pointed to by *str* to a type **unsigned long** and **unsigned long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a `'+'` or `'−'` sign. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *str* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer. |

CX The *strtoul*( ) function shall not change the setting of *errno* if successful.

Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *strtoul*( ) or *strtoull*( ), then check *errno*. |

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned CX and *errno* may be set to [EINVAL]. If the correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX} shall be returned and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [EINVAL]	The value of *base* is not supported.
[ERANGE]	The value to be returned is not representable.

These functions may fail if:

CX [EINVAL]	No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isalpha( ), *scanf*( ), *strtod*( ), *strtol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

  In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *strtoul*( ) prototype is updated.
- The *strtoull*( ) function is added.

NAME

strtoumax — convert string to integer type

SYNOPSIS

```
#include <inttypes.h>

uintmax_t strtoumax(const char *restrict nptr, char **restrict endptr,
    int base);
```

DESCRIPTION

Refer to *strtoimax*( ).

NAME

strxfrm — string transformation

SYNOPSIS

```
#include <string.h>

size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *strxfrm*( ) function shall transform the string pointed to by *s2* and place the resulting string into the array pointed to by *s1*. The transformation is such that if *strcmp*( ) is applied to two transformed strings, it shall return a value greater than, equal to, or less than 0, corresponding to the result of *strcoll*( ) applied to the same two original strings. No more than *n* bytes are placed into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is 0, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

CX The *strxfrm*( ) function shall not change the setting of *errno* if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *strxfrm*( ), then check *errno*. |

RETURN VALUE

Upon successful completion, *strxfrm*( ) shall return the length of the transformed string (not including the terminating null byte). If the value returned is *n* or more, the contents of the array pointed to by *s1* are unspecified. |

CX On error, *strxfrm*( ) may set *errno* but no return value is reserved to indicate an error.

ERRORS

The *strxfrm*( ) function may fail if:

CX [EINVAL] The string pointed to by the *s2* argument contains characters outside the domain of the collating sequence.

EXAMPLES

None.

APPLICATION USAGE

The transformation function is such that two transformed strings can be ordered by *strcmp*( ) as appropriate to collating sequence information in the program's locale (category *LC_COLLATE*).

The fact that when *n* is 0 *s1* is permitted to be a null pointer is useful to determine the size of the *s1* array prior to making the transformation.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strcmp( ), *strcoll*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<string.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the ISO C standard.

Issue 5

The DESCRIPTION is updated to indicate that *errno* does not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The *strxfrm*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

swab — swap bytes

SYNOPSIS

XSI

```
#include <unistd.h>

void swab(const void *restrict src, void *restrict dest,
    ssize_t nbytes);
```

DESCRIPTION

The *swab*( ) function shall copy *nbytes* bytes, which are pointed to by *src*, to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, *swab*( ) copies and exchanges *nbytes*−1 bytes and the disposition of the last byte is unspecified. If copying takes place between objects that overlap, the behavior is undefined. If *nbytes* is negative, *swab*( ) does nothing.

RETURN VALUE

None.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The **restrict** keyword is added to the *swab*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

swapcontext — swap user context

SYNOPSIS

XSI

```
#include <ucontext.h>

int swapcontext(ucontext_t *restrict oucp,
    const ucontext_t *restrict ucp);
```

DESCRIPTION

Refer to *makecontext*( ).

NAME

swprintf — print formatted wide-character output

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int swprintf(wchar_t *ws, size_t n, const wchar_t *format, ...);
```

DESCRIPTION

Refer to *fwprintf*( ).

NAME

swscanf — convert formatted wide-character input

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int swscanf(const wchar_t *restrict ws,
    const wchar_t *restrict format, ... );
```

DESCRIPTION

Refer to *fwscanf*( ).

NAME

symlink — make symbolic link to a file

SYNOPSIS

```
#include <unistd.h>

int symlink(const char *path1, const char *path2);
```

DESCRIPTION

The *symlink*( ) function shall create a symbolic link called *path2* that contains the string pointed to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the symbolic link).

The string pointed to by *path1* shall be treated only as a character string and shall not be validated as a pathname.

If the *symlink*( ) function fails for any reason other than [EIO], any file named by *path2* shall be unaffected.

RETURN VALUE

Upon successful completion, *symlink*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *symlink*( ) function shall fail if:

[EACCES]	Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the path prefix of *path2*.
[EEXIST]	The *path2* argument names an existing file or symbolic link.
[EIO]	An I/O error occurs while reading from or writing to the file system.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path2* argument.
[ENAMETOOLONG]	The length of the *path2* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} or the length of the *path1* argument is longer than {SYMLINK_MAX}.
[ENOENT]	A component of *path2* does not name an existing file or *path2* is an empty string.
[ENOSPC]	The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory, or the new symbolic link cannot be created because no space is left on the file system which shall contain the link, or the file system is out of file-allocation resources.
[ENOTDIR]	A component of the path prefix of *path2* is not a directory.
[EROFS]	The new symbolic link would reside on a read-only file system.

The *symlink*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path2* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path2*

argument, the length of the substituted pathname string exceeded {PATH_MAX} bytes (including the terminating null byte), or the length of the string pointed to by *path1* exceeded {SYMLINK_MAX}.

EXAMPLES

None.

APPLICATION USAGE

Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance; in fact, the file named by the *path1* argument need not exist when the link is created. A symbolic link can cross file system boundaries.

Normal permission checks are made on each component of the symbolic link pathname during its resolution.

RATIONALE

Since IEEE Std 1003.1-200x does not require any association of file times with symbolic links, there is no requirement that file times be updated by *symlink*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

lchown( ), *link*( ), *lstat*( ), *open*( ), *readlink*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION text is updated.
- The [ELOOP] optional error condition is added.

NAME

sync — schedule file system updates

SYNOPSIS

XSI

```
#include <unistd.h>

void sync(void);
```

DESCRIPTION

The *sync*( ) function shall cause all information in memory that updates file systems to be scheduled for writing out to all file systems.

The writing, although scheduled, is not necessarily complete upon return from *sync*( ).

RETURN VALUE

The *sync*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fsync( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

NAME

sysconf — get configurable system variables

SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);
```

DESCRIPTION

The *sysconf*( ) function provides a method for the application to determine the current value of a configurable system limit or option (*variable*). Support for some system variables is dependent on implementation options (as indicated by the margin codes in the following table). Where an implementation option is not supported, the variable need not be supported.

The *name* argument represents the system variable to be queried. The following table lists the minimal set of system variables from **<limits.h>** or **<unistd.h>** that can be returned by *sysconf*( ), and the symbolic constants, defined in **<unistd.h>** that are the corresponding values used for *name*. Support for some configuration variables is dependent on implementation options (see shading and margin codes in the table below). Where an implementation option is not supported, the variable need not be supported.

	Variable	**Value of Name**
AIO	{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
	{AIO_MAX}	_SC_AIO_MAX
	{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
	{ARG_MAX}	_SC_ARG_MAX
XSI	{ATEXIT_MAX}	_SC_ATEXIT_MAX
	{BC_BASE_MAX}	_SC_BC_BASE_MAX
	{BC_DIM_MAX}	_SC_BC_DIM_MAX
	{BC_SCALE_MAX}	_SC_BC_SCALE_MAX
	{BC_STRING_MAX}	_SC_BC_STRING_MAX
	{CHILD_MAX}	_SC_CHILD_MAX
	Clock ticks/second	_SC_CLK_TCK
	{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
XSI	{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
	{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
	{HOST_NAME_MAX}	_SC_HOST_NAME_MAX
XSI	{IOV_MAX}	_SC_IOV_MAX
	{LINE_MAX}	_SC_LINE_MAX
	{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
	{NGROUPS_MAX}	_SC_NGROUPS_MAX
TSF	Maximum size of *getgrgid_r*( ) and *getgrnam_r*( ) data buffers	_SC_GETGR_R_SIZE_MAX
	Maximum size of *getpwuid_r*( ) and *getpwnam_r*( ) data buffers	_SC_GETPW_R_SIZE_MAX
MSG	{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
	{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
	{OPEN_MAX}	_SC_OPEN_MAX

	Variable	Value of Name
ADV	_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
BAR	_POSIX_BARRIERS	_SC_BARRIERS
AIO	_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
	_POSIX_BASE	_SC_BASE
	_POSIX_C_LANG_SUPPORT	_SC_C_LANG_SUPPORT
	_POSIX_C_LANG_SUPPORT_R	_SC_C_LANG_SUPPORT_R
CS	_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION
CPT	_POSIX_CPUTIME	_SC_CPUTIME
	_POSIX_DEVICE_IO	_SC_DEVICE_IO
	_POSIX_DEVICE_SPECIFIC	_SC_DEVICE_SPECIFIC
	_POSIX_DEVICE_SPECIFIC_R	_SC_DEVICE_SPECIFIC_R
	_POSIX_FD_MGMT	_SC_FD_MGMT
	_POSIX_FIFO	_SC_FIFO
	_POSIX_FILE_ATTRIBUTES	_SC_FILE_ATTRIBUTES
	_POSIX_FILE_LOCKING	_SC_FILE_LOCKING
	_POSIX_FILE_SYSTEM	_SC_FILE_SYSTEM
FSC	_POSIX_FSYNC	_SC_FSYNC
	_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
MF	_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
ML	_POSIX_MEMLOCK	_SC_MEMLOCK
MLR	_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
MPR	_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
MSG	_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
MON	_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
	_POSIX_MULTI_PROCESS	_SC_MULTI_PROCESS
	_POSIX_NETWORKING	_SC_NETWORKING
	_POSIX_PIPE	_SC_PIPE
PIO	_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
PS	_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
THR	_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
RTS	_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
	_POSIX_REGEXP	_SC_REGEXP
	_POSIX_SAVED_IDS	_SC_SAVED_IDS
SEM	_POSIX_SEMAPHORES	_SC_SEMAPHORES
SHM	_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS
	_POSIX_SHELL	_SC_SHELL
	_POSIX_SIGNALS	_SC_SIGNALS
	_POSIX_SINGLE_PROCESS	_SC_SINGLE_PROCESS
SPN	_POSIX_SPAWN	_SC_SPAWN
SPI	_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
SS	_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
SIO	_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
	_POSIX_SYSTEM_DATABASE	_SC_SYSTEM_DATABASE
	_POSIX_SYSTEM_DATABASE_R	_SC_SYSTEM_DATABASE_R

	Variable	Value of Name
TSA	_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
TSS	_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
TCT	_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
TPI	_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
TPP	_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
TPS	_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
TSH	_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
TSF	_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
TSP	_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
THR	_POSIX_THREADS	_SC_THREADS
TMO	_POSIX_TIMEOUTS	_SC_TIMEOUTS
TMR	_POSIX_TIMERS	_SC_TIMERS
TRC	_POSIX_TRACE	_SC_TRACE
TEF	_POSIX_TRACE_EVENT_FILTER	_SC_TRACE_EVENT_FILTER
TRI	_POSIX_TRACE_INHERIT	_SC_TRACE_INHERIT
TRL	_POSIX_TRACE_LOG	_SC_TRACE_LOG
TYM	_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
	_POSIX_USER_GROUPS	_SC_USER_GROUPS
	_POSIX_USER_GROUPS_R	_SC_USER_GROUPS_R
	_POSIX_VERSION	_SC_VERSION
	_POSIX_V6_ILP32_OFF32	_SC_V6_ILP32_OFF32
	_POSIX_V6_ILP32_OFFBIG	_SC_V6_ILP32_OFFBIG
	_POSIX_V6_LP64_OFF64	_SC_V6_LP64_OFF64
	_POSIX_V6_LPBIG_OFFBIG	_SC_V6_LPBIG_OFFBIG
	_POSIX2_C_BIND	_SC_2_C_BIND
	_POSIX2_C_DEV	_SC_2_C_DEV
	_POSIX2_C_VERSION	_SC_2_C_VERSION
	_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
	_POSIX2_FORT_DEV	_SC_2_FORT_DEV
	_POSIX2_FORT_RUN	_SC_2_FORT_RUN
	_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
BE	_POSIX2_PBS	_SC_2_PBS
	_POSIX2_PBS_ACCOUNTING	_SC_2_PBS_ACCOUNTING
	_POSIX2_PBS_LOCATE	_SC_2_PBS_LOCATE
	_POSIX2_PBS_MESSAGE	_SC_2_PBS_MESSAGE
	_POSIX2_PBS_TRACK	_SC_2_PBS_TRACK
	_POSIX2_SW_DEV	_SC_2_SW_DEV
	_POSIX2_UPE	_SC_2_UPE
	_POSIX2_VERSION	_SC_2_VERSION
	_REGEX_VERSION	_SC_REGEX_VERSION
XSI	{PAGE_SIZE}	_SC_PAGE_SIZE
	{PAGESIZE}	_SC_PAGESIZE

	Variable	Value of Name
THR	{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
	{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
	{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
	{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
	{RE_DUP_MAX}	_SC_RE_DUP_MAX
RTS	{RTSIG_MAX}	_SC_RTSIG_MAX
SEM	{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
	{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
RTS	{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
	{STREAM_MAX}	_SC_STREAM_MAX
	{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
TMR	{TIMER_MAX}	_SC_TIMER_MAX
	{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
	{TZNAME_MAX}	_SC_TZNAME_MAX
XSI	_XBS5_ILP32_OFF32 (**LEGACY**)	_SC_XBS5_ILP32_OFF32 (**LEGACY**)
	_XBS5_ILP32_OFFBIG (**LEGACY**)	_SC_XBS5_ILP32_OFFBIG (**LEGACY**)
	_XBS5_LP64_OFF64 (**LEGACY**)	_SC_XBS5_LP64_OFF64 (**LEGACY**)
	_XBS5_LPBIG_OFFBIG (**LEGACY**)	_SC_XBS5_LPBIG_OFFBIG (**LEGACY**)
	_XOPEN_CRYPT	_SC_XOPEN_CRYPT
	_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N
	_XOPEN_LEGACY	_SC_XOPEN_LEGACY
	_XOPEN_REALTIME	_SC_XOPEN_REALTIME
	_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
	_XOPEN_SHM	_SC_XOPEN_SHM
	_XOPEN_UNIX	_SC_XOPEN_UNIX
	_XOPEN_VERSION	_SC_XOPEN_VERSION
	_XOPEN_XCU_VERSION	_SC_XOPEN_XCU_VERSION

RETURN VALUE

If *name* is an invalid value, *sysconf*( ) shall return −1 and set *errno* to indicate the error. If the variable corresponding to *name* has no limit, *sysconf*( ) shall return −1 without changing the value of *errno*. Note that indefinite limits do not imply infinite limits; see **<limits.h>**.

Otherwise, *sysconf*( ) shall return the current variable value on the system. The value returned shall not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's **<limits.h>** or **<unistd.h>**. The value shall not change during the lifetime of the calling process.

ERRORS

The *sysconf*( ) function shall fail if:

[EINVAL] The value of the *name* argument is invalid.

EXAMPLES

None.

APPLICATION USAGE

As −1 is a permissible return value in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *sysconf*( ), and, if it returns −1, check to see if *errno* is non-zero.

If the value of *sysconf*(_SC_2_VERSION) is not equal to the value of the _POSIX2_VERSION symbolic constant, the utilities available via *system*( ) or *popen*( ) might not behave as described in the Shell and Utilities volume of IEEE Std 1003.1-200x. This would mean that the application is not running in an environment that conforms to the Shell and Utilities volume of IEEE Std 1003.1-200x. Some applications might be able to deal with this, others might not. However, the functions defined in this volume of IEEE Std 1003.1-200x continue to operate as specified, even if: *sysconf*(_SC_2_VERSION) reports that the utilities no longer perform as specified.

RATIONALE

This functionality was added in response to requirements of application developers and of system vendors who deal with many international system configurations. It is closely related to *pathconf*( ) and *fpathconf*( ).

Although a conforming application can run on all systems by never demanding more resources than the minimum values published in this volume of IEEE Std 1003.1-200x, it is useful for that application to be able to use the actual value for the quantity of a resource available on any given system. To do this, the application makes use of the value of a symbolic constant in **<limits.h>** or **<unistd.h>**.

However, once compiled, the application must still be able to cope if the amount of resource available is increased. To that end, an application may need a means of determining the quantity of a resource, or the presence of an option, at execution time.

Two examples are offered:

1. Applications may wish to act differently on systems with or without job control. Applications vendors who wish to distribute only a single binary package to all instances of a computer architecture would be forced to assume job control is never available if it were to rely solely on the **<unistd.h>** value published in this volume of IEEE Std 1003.1-200x.
2. International applications vendors occasionally require knowledge of the number of clock ticks per second. Without these facilities, they would be required to either distribute their applications partially in source form or to have 50Hz and 60Hz versions for the various countries in which they operate.

It is the knowledge that many applications are actually distributed widely in executable form that leads to this facility. If limited to the most restrictive values in the headers, such applications would have to be prepared to accept the most limited environments offered by the smallest microcomputers. Although this is entirely portable, there was a consensus that they should be able to take advantage of the facilities offered by large systems, without the restrictions associated with source and object distributions.

During the discussions of this feature, it was pointed out that it is almost always possible for an application to discern what a value might be at runtime by suitably testing the various functions themselves. And, in any event, it could always be written to adequately deal with error returns from the various functions. In the end, it was felt that this imposed an unreasonable level of complication and sophistication on the application writer.

This runtime facility is not meant to provide ever-changing values that applications have to check multiple times. The values are seen as changing no more frequently than once per system initialization, such as by a system administrator or operator with an automatic configuration program. This volume of IEEE Std 1003.1-200x specifies that they shall not change within the lifetime of the process.

Some values apply to the system overall and others vary at the file system or directory level. The latter are described in *pathconf*( ).

Note that all values returned must be expressible as integers. String values were considered, but the additional flexibility of this approach was rejected due to its added complexity of implementation and use.

Some values, such as {PATH_MAX}, are sometimes so large that they must not be used to, say, allocate arrays. The *sysconf*( ) function returns a negative value to show that this symbolic constant is not even defined in this case.

Similar to *pathconf*( ), this permits the implementation not to have a limit. When one resource is infinite, returning an error indicating that some other resource limit has been reached is conforming behavior.

FUTURE DIRECTIONS

None.

SEE ALSO

confstr( ), *pathconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<unistd.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *getconf*

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

The _XBS_ variables and name values are added to the table of system variables in the DESCRIPTION. These are all marked EX.

Issue 6

The symbol CLK_TCK is obsolescent and removed. It is replaced with the phrase ‘‘clock ticks per second’’.

The symbol {PASS_MAX} is removed.

The following changes were made to align with the IEEE P1003.1a draft standard:

- Table entries added for the following variables: _SC_REGEXP, _SC_SHELL, _SC_REGEX_VERSION, _SC_SYMLOOP_MAX.

The following *sysconf*( ) variables and their associated names are added for alignment with IEEE Std 1003.1d-1999:

_POSIX_ADVISORY_INFO
_POSIX_CPUTIME
_POSIX_SPAWN
_POSIX_SPORADIC_SERVER
_POSIX_THREAD_CPUTIME
_POSIX_THREAD_SPORADIC_SERVER
_POSIX_TIMEOUTS

The following changes are made to the DESCRIPTION for alignment with IEEE Std 1003.1j-2000:

- A statement expressing the dependency of support for some system variables on implementation options is added.

- The following system variables are added:

_POSIX_BARRIERS
_POSIX_CLOCK_SELECTION
_POSIX_MONOTONIC_CLOCK
_POSIX_READER_WRITER_LOCKS
_POSIX_SPIN_LOCKS
_POSIX_TYPED_MEMORY_OBJECTS

The following system variables are added for alignment with IEEE Std 1003.2d-1994:

_POSIX2_PBS
_POSIX2_PBS_ACCOUNTING
_POSIX2_PBS_LOCATE
_POSIX2_PBS_MESSAGE
_POSIX2_PBS_TRACK

The following *sysconf*( ) variables and their associated names are added for alignment with IEEE Std 1003.1q-2000:

_POSIX_TRACE
_POSIX_TRACE_EVENT_FILTER
_POSIX_TRACE_INHERIT
_POSIX_TRACE_LOG

The macros associated with the *c89* programming models are marked LEGACY, and new equivalent macros associated with *c99* are introduced.

NAME

syslog — log a message

SYNOPSIS

XSI

```
#include <syslog.h>

void syslog(int priority, const char *message, ... /* argument */);
```

DESCRIPTION

Refer to *closelog*( ).

NAME

system — issue a command

SYNOPSIS

```
#include <stdlib.h>

int system(const char *command);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *command* is a null pointer, the *system*( ) function shall determine whether the host environment has a command processor. If *command* is not a null pointer, the *system*( ) function shall pass the string pointed to by *command* to that command processor to be executed in an implementation-defined manner; this might then cause the program calling *system*( ) to behave in a non-conforming manner or to terminate.

CX The environment of the executed command shall be as if a child process were created using *fork*( ), and the child process invoked the *sh* utility using *execl*( ) as follows:

```
execl(<shell path>, "sh", "-c", command, (char *)0);
```

where <*shell path*> is an unspecified pathname for the *sh* utility.

The *system*( ) function shall ignore the SIGINT and SIGQUIT signals, and shall block the SIGCHLD signal, while waiting for the command to terminate. If this might cause the application to miss a signal that would have killed it, then the application should examine the return value from *system*( ) and take whatever action is appropriate to the application if the command terminated due to receipt of a signal.

The *system*( ) function shall not affect the termination status of any child of the calling processes other than the process or processes it itself creates.

The *system*( ) function shall not return until the child process has terminated.

RETURN VALUE

If *command* is a null pointer, *system*( ) shall return non-zero to indicate that a command processor is available, or zero if none is available. CX The *system*( ) function shall always return non-zero when *command* is NULL.

CX If *command* is not a null pointer, *system*( ) shall return the termination status of the command language interpreter in the format specified by *waitpid*( ). The termination status shall be as defined for the *sh* utility; otherwise, the termination status is unspecified. If some error prevents the command language interpreter from executing after the child process is created, the return value from *system*( ) shall be as if the command language interpreter had terminated using *exit*(127) or *_exit*(127). If a child process cannot be created, or if the termination status for the command language interpreter cannot be obtained, *system*( ) shall return −1 and set *errno* to indicate the error.

ERRORS

CX The *system*( ) function may set *errno* values as described by *fork*( ).

In addition, *system*( ) may fail if:

CX [ECHILD] The status of the child process created by *system*( ) is no longer available.

EXAMPLES

None.

APPLICATION USAGE

If the return value of *system*( ) is not –1, its value can be decoded through the use of the macros described in **<sys/wait.h>**. For convenience, these macros are also provided in **<stdlib.h>**.

Note that, while *system*( ) must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork*( ) and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system*( ) is called, then the child is started with SIGINT handling set to SIG_DFL.

Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the `'!'` command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses *system*( ) to perform some task invisible to the user. If the user typed the interrupt character (`"^C"`, for example) while *system*( ) is being used in this way, one would expect the application to be killed, but only the executed command is killed. Applications that use *system*( ) in this way should carefully check the return status from *system*( ) to see if the executed command was successful, and should take appropriate action when the command fails.

Blocking SIGCHLD while waiting for the child to terminate prevents the application from catching the signal and obtaining status from *system*( )'s child process before *system*( ) can get the status itself.

The context in which the utility is ultimately executed may differ from that in which *system*( ) was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the process ID and parent process ID are different. Also, if the executed utility changes its environment variables or its current working directory, that change is not reflected in the caller's context.

There is no defined way for an application to find the specific path for the shell. However, *confstr*( ) can provide a value for *PATH* that is guaranteed to find the *sh* utility.

RATIONALE

The *system*( ) function should not be used by programs that have set user (or group) ID privileges. The *fork*( ) and *exec* family of functions (except *execlp*( ) and *execvp*( )), should be used instead. This prevents any unforeseen manipulation of the environment of the user that could cause execution of commands not anticipated by the calling program.

There are three levels of specification for the *system*( ) function. The ISO C standard gives the most basic. It requires that the function exists, and defines a way for an application to query whether a command language interpreter exists. It says nothing about the command language or the environment in which the command is interpreted.

IEEE Std 1003.1-200x places additional restrictions on *system*( ). It requires that if there is a command language interpreter, the environment must be as specified by *fork*( ) and *exec*. This ensures, for example, that close-on-*exec* works, that file locks are not inherited, and that the process ID is different. It also specifies the return value from *system*( ) when the command line can be run, thus giving the application some information about the command's completion statu.

Finally, IEEE Std 1003.1-200x requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of IEEE Std 1003.1-200x.

Note that, *system*(NULL) is required to return non-zero, indicating that there is a command language interpreter. At first glance, this would seem to conflict with the ISO C standard which allows *system*(NULL) to return zero. There is no conflict, however. A system must have a command language interpreter, and is non-conforming if none is present. It is therefore permissible for the *system*( ) function on such a system to implement the behavior specified by the ISO C standard as long as it is understood that the implementation does not conform to IEEE Std 1003.1-200x if *system*(NULL) returns zero.

It was explicitly decided that when *command* is NULL, *system*( ) should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, and so on. The call *system*(NULL) could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause any *system*( ) call to fail, that system has been rendered non-conforming.

Early drafts required, or allowed, *system*( ) to return with *errno* set to [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that *system*( ) not return until the child has terminated was added. This means that if a *waitpid*( ) call in *system*( ) exits with *errno* set to [EINTR], *system*( ) must re-issue the *waitpid*( ). This change was made for two reasons:

1. There is no way for an application to clean up if *system*( ) returns [EINTR], short of calling *wait*( ), and that could have the undesirable effect of returning the status of children other than the one started by *system*( ).

2. While it might require a change in some historical implementations, those implementations already have to be changed because they use *wait*( ) instead of *waitpid*( ).

Note that if the application is catching SIGCHLD signals, it will receive such a signal before a successful *system*( ) call returns.

To conform to IEEE Std 1003.1-200x, *system*( ) must use *waitpid*( ), or some similar function, instead of *wait*( ).

The following code sample illustrates how *system*( ) might be implemented on an implementation conforming to IEEE Std 1003.1-200x.

```
#include <signal.h>
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;
    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
    sigemptyset(&savequit.sa_mask);
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
```

```
        sigaddset(&sa.sa_mask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
        if ((pid = fork()) == 0) {
            sigaction(SIGINT, &savintr, (struct sigaction *)0);
            sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
            sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
            execl("/bin/sh", "sh", "-c", cmd, (char *)0);
            _exit(127);
        }
        if (pid == -1) {
            stat = -1; /* errno comes from fork() */
        } else {
            while (waitpid(pid, &stat, 0) == -1) {
                if (errno != EINTR){
                    stat = -1;
                    break;
                }
            }
        }
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        return(stat);
    }
```

Note that, while a particular implementation of *system*( ) (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not portable, and is not intended to be.

One reviewer suggested that an implementation of *system*( ) might want to use an environment variable such as *SHELL* to determine which command interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using *system*( ), to specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, then changing *SHELL* would render *system*( ) non-conforming. This would affect applications that expected the specified behavior from *system*( ), and since the Shell and Utilities volume of IEEE Std 1003.1-200x does not mention that *SHELL* affects *system*( ), the application would not know that it needed to unset *SHELL*.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *pipe*( ), *waitpid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<signal.h>**, **<stdlib.h>**, **<sys/wait.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x, *sh* |

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID. |

Issue 6

The following changes were made to align with the IEEE P1003.1a draft standard:

- The DESCRIPTION is adjusted to reflect the behavior on systems that do not support the Shell option.

NAME

tan, tanf, tanl — tangent function

SYNOPSIS

```
#include <math.h>

double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the tangent of their argument *x*, measured in radians.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the tangent of *x*.

If the correct value would cause underflow, and is not representable, a range error may occur, and MX either 0.0 (if supported), or an implementation-defined value shall be returned.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

If *x* is ±Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value shall be returned.

XSI If the correct value would cause overflow, a range error shall occur and *tan*( ), *tanf*( ), and *tanl*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, and HUGE_VALL, respectively.

ERRORS

These functions shall fail if:

MX	Domain Error	The value *x* is ±Inf.
		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.
XSI	Range Error	The result overflows
		If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

These functions may fail if:

MX Range Error The result underflows, or the value *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

Taking the Tangent of a 45-Degree Angle

```
#include <math.h>
...
double radians = 45.0 * M_PI / 180;
double result;
...
result = tan (radians);
```

APPLICATION USAGE

There are no known floating-point representations such that for a normal argument, *tan*(x) is either overflow or underflow.

These functions may lose accuracy when their argument is near a multiple of $\pi/2$ or is far from 0.0.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atan( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

The *tanf*( ) and *tanl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

tanf — tangent function

SYNOPSIS

```
#include <math.h>

float tanf(float x);
```

DESCRIPTION

Refer to *tan*( ).

NAME

tanh, tanhf, tanhl — hyperbolic tangent functions

SYNOPSIS

```
#include <math.h>

double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the hyperbolic tangent of their argument *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the hyperbolic tangent of *x*.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, *x* shall be returned.

If *x* is ±Inf, ±1 shall be returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

ERRORS

These functions may fail if:

MX Range Error The value of *x* is subnormal.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

atanh( ), *feclearexcept*( ), *fetestexcept*( ), *isnan*( ), *tan*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The *tanhf*( ) and *tanhl*( ) functions are added for alignment with the ISO/IEC 9899:1999 standard.

The DESCRIPTION, RETURN VALUE, ERRORS, and APPLICATION USAGE sections are revised to align with the ISO/IEC 9899:1999 standard.

IEC 60559:1989 standard floating-point extensions over the ISO/IEC 9899:1999 standard are marked.

NAME

tanl — tangent function

SYNOPSIS

```
#include <math.h>

long double tanl(long double x);
```

DESCRIPTION

Refer to *tan*( ).

NAME

tcdrain — wait for transmission of output

SYNOPSIS

```
#include <termios.h>

int tcdrain(int fildes);
```

DESCRIPTION

The *tcdrain*( ) function shall block until all output written to the object referred to by *fildes* is transmitted. The *fildes* argument is an open file descriptor associated with a terminal.

Any attempts to use *tcdrain*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcdrain*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINTR]	A signal interrupted *tcdrain*( ).
[ENOTTY]	The file associated with *fildes* is not a terminal.

The *tcdrain*( ) function may fail if:

[EIO]	The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcflush( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the final paragraph is no longer conditional on _POSIX_JOB_CONTROL. This is a FIPS requirement.

- The [EIO] error is added.

NAME

tcflow — suspend or restart the transmission or reception of data

SYNOPSIS

```
#include <termios.h>

int tcflow(int fildes, int action);
```

DESCRIPTION

The *tcflow*( ) function shall suspend or restart transmission or reception of data on the object referred to by *fildes*, depending on the value of *action*. The *fildes* argument is an open file descriptor associated with a terminal.

- If *action* is TCOOFF, output shall be suspended.
- If *action* is TCOON, suspended output shall be restarted.
- If *action* is TCIOFF, the system shall transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system.
- If *action* is TCION, the system shall transmit a START character, which is intended to cause the terminal device to start transmitting data to the system.

The default on the opening of a terminal file is that neither its input nor its output are suspended.

Attempts to use *tcflow*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal, shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcflow*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *action* argument is not a supported value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

The *tcflow*( ) function may fail if:

[EIO]	The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcsendbreak( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EIO] error is added.

NAME

tcflush — flush non-transmitted output data, non-read input data, or both

SYNOPSIS

```
#include <termios.h>

int tcflush(int fildes, int queue_selector);
```

DESCRIPTION

Upon successful completion, *tcflush*( ) shall discard data written to the object referred to by *fildes* (an open file descriptor associated with a terminal) but not transmitted, or data received but not read, depending on the value of *queue_selector*:

- If *queue_selector* is TCIFLUSH, it shall flush data received but not read.
- If *queue_selector* is TCOFLUSH, it shall flush data written but not transmitted.
- If *queue_selector* is TCIOFLUSH, it shall flush both data received but not read and data written but not transmitted.

Attempts to use *tcflush*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcflush*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	The *queue_selector* argument is not a supported value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

The *tcflush*( ) function may fail if:

[EIO]	The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcdrain( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The Open Group Corrigendum U035/1 is applied. In the ERRORS and APPLICATION USAGE sections, references to *tcflow*( ) are replaced with *tcflush*( ).

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the final paragraph is no longer conditional on _POSIX_JOB_CONTROL. This is a FIPS requirement.
- The [EIO] error is added.

NAME

tcgetattr — get the parameters associated with the terminal

SYNOPSIS

```
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);
```

DESCRIPTION

The *tcgetattr*( ) function shall get the parameters associated with the terminal referred to by *fildes* and store them in the **termios** structure referenced by *termios_p*. The *fildes* argument is an open file descriptor associated with a terminal.

The *termios_p* argument is a pointer to a **termios** structure.

The *tcgetattr*( ) operation is allowed from any process.

If the terminal device supports different input and output baud rates, the baud rates stored in the **termios** structure returned by *tcgetattr*( ) shall reflect the actual baud rates, even if they are equal. If differing baud rates are not supported, the rate returned as the output baud rate shall be the actual baud rate. If the terminal device does not support split baud rates, the input baud rate stored in the **termios** structure shall be the output rate (as one of the symbolic values).

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetattr*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The file associated with *fildes* is not a terminal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Care must be taken when changing the terminal attributes. Applications should always do a *tcgetattr*( ), save the **termios** structure values returned, and then do a *tcsetattr*( ) changing only the necessary fields. The application should use the values saved from the *tcgetattr*( ) to reset the terminal state whenever it is done with the terminal. This is necessary because terminal attributes apply to the underlying port and not to each individual open instance; that is, all processes that have used the terminal see the latest attribute changes.

A program that uses these functions should be written to catch all signals and take other appropriate actions to ensure that when the program terminates, whether planned or not, the terminal device's state is restored to its original state.

Existing practice dealing with error returns when only part of a request can be honored is based on calls to the *ioctl*( ) function. In historical BSD and System V implementations, the corresponding *ioctl*( ) returns zero if the requested actions were semantically correct, even if some of the requested changes could not be made. Many existing applications assume this behavior and would no longer work correctly if the return value were changed from zero to −1 in this case.

Note that either specification has a problem. When zero is returned, it implies everything succeeded even if some of the changes were not made. When −1 is returned, it implies everything failed even though some of the changes were made.

Applications that need all of the requested changes made to work properly should follow *tcsetattr*( ) with a call to *tcgetattr*( ) and compare the appropriate field values.

FUTURE DIRECTIONS

None.

SEE ALSO

tcsetattr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the DESCRIPTION, the rate returned as the input baud rate shall be the output rate. Previously, the number zero was also allowed but was obsolescent.

NAME

tcgetpgrp — get the foreground process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t tcgetpgrp(int fildes);
```

DESCRIPTION

The *tcgetpgrp*( ) function shall return the value of the process group ID of the foreground process group associated with the terminal.

If there is no foreground process group, *tcgetpgrp*( ) shall return a value greater than 1 that does not match the process group ID of any existing process group.

The *tcgetpgrp*( ) function is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

RETURN VALUE

Upon successful completion, *tcgetpgrp*( ) shall return the value of the process group ID of the foreground process associated with the terminal. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetpgrp*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setsid( ), *setpgid*( ), *tcsetpgrp*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

- In the DESCRIPTION, text previously conditional on support for _POSIX_JOB_CONTROL is now mandatory. This is a FIPS requirement.

NAME

tcgetsid — get process group ID for session leader for controlling terminal

SYNOPSIS

XSI

```
#include <termios.h>

pid_t tcgetsid(int fildes);
```

DESCRIPTION

The *tcgetsid*( ) function shall obtain the process group ID of the session for which the terminal specified by *fildes* is the controlling terminal.

RETURN VALUE

Upon successful completion, *tcgetsid*( ) shall return the process group ID associated with the terminal. Otherwise, a value of (**pid_t**)–1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcgetsid*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The calling process does not have a controlling terminal, or the file is not the controlling terminal.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The [EACCES] error has been removed from the list of mandatory errors, and the description of [ENOTTY] has been reworded.

NAME

tcsendbreak — send a ‘‘break’’ for a specific duration

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

DESCRIPTION

If the terminal is using asynchronous serial data transmission, *tcsendbreak*( ) shall cause transmission of a continuous stream of zero-valued bits for a specific duration. If *duration* is 0, it shall cause transmission of zero-valued bits for at least 0,25 seconds, and not more than 0,5 seconds. If *duration* is not 0, it shall send zero-valued bits for an implementation-defined period of time.

The *fildes* argument is an open file descriptor associated with a terminal.

If the terminal is not using asynchronous serial data transmission, it is implementation-defined whether *tcsendbreak*( ) sends data to generate a break condition or returns without taking any action.

Attempts to use *tcsendbreak*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcsendbreak*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The file associated with *fildes* is not a terminal.

The *tcsendbreak*( ) function may fail if:

[EIO]	The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.
- The [EIO] error is added.

NAME

tcsetattr — set the parameters associated with the terminal

SYNOPSIS

```
#include <termios.h>

int tcsetattr(int fildes, int optional_actions,
    const struct termios *termios_p);
```

DESCRIPTION

The *tcsetattr*( ) function shall set the parameters associated with the terminal referred to by the open file descriptor *fildes* (an open file descriptor associated with a terminal) from the **termios** structure referenced by *termios_p* as follows:

- If *optional_actions* is TCSANOW, the change shall occur immediately.
- If *optional_actions* is TCSADRAIN, the change shall occur after all output written to *fildes* is transmitted. This function should be used when changing parameters that affect output.
- If *optional_actions* is TCSAFLUSH, the change shall occur after all output written to *fildes* is transmitted, and all input so far received but not read shall be discarded before the change is made.

If the output baud rate stored in the **termios** structure pointed to by *termios_p* is the zero baud rate, B0, the modem control lines shall no longer be asserted. Normally, this shall disconnect the line.

If the input baud rate stored in the **termios** structure pointed to by *termios_p* is 0, the input baud rate given to the hardware is the same as the output baud rate stored in the **termios** structure.

The *tcsetattr*( ) function shall return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It shall set all the attributes that the implementation supports as requested and leaves all the attributes not supported by the implementation unchanged. If no part of the request can be honored, it shall return −1 and set *errno* to [EINVAL]. If the input and output baud rates differ and are a combination that is not supported, neither baud rate shall be changed. A subsequent call to *tcgetattr*( ) shall return the actual state of the terminal device (reflecting both the changes made and not made in the previous *tcsetattr*( ) call). The *tcsetattr*( ) function shall not change the values found in the **termios** structure under any circumstances.

The effect of *tcsetattr*( ) is undefined if the value of the **termios** structure pointed to by *termios_p* was not derived from the result of a call to *tcgetattr*( ) on *fildes*; an application should modify only fields and flags defined by this volume of IEEE Std 1003.1-200x between the call to *tcgetattr*( ) and *tcsetattr*( ), leaving all other fields and flags unmodified.

No actions defined by this volume of IEEE Std 1003.1-200x, other than a call to *tcsetattr*( ) or a close of the last file descriptor in the system associated with this terminal device, shall cause any of the terminal attributes defined by this volume of IEEE Std 1003.1-200x to change.

If *tcsetattr*( ) is called from a process which is a member of a background process group on a *fildes* associated with its controlling terminal:

- If the calling process is blocking or ignoring SIGTTOU signals, the operation completes normally and no signal is sent.
- Otherwise, a SIGTTOU signal shall be sent to the process group.

RETURN VALUE

Upon successful completion, 0 shall be shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error. |

ERRORS

The *tcsetattr*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINTR]	A signal interrupted *tcsetattr*( ).
[EINVAL]	The *optional_actions* argument is not a supported value, or an attempt was made to change an attribute represented in the **termios** structure to an unsupported value.
[ENOTTY]	The file associated with *fildes* is not a terminal.

The *tcsetattr*( ) function may fail if:

[EIO]	The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

EXAMPLES

None.

APPLICATION USAGE

If trying to change baud rates, applications should call *tcsetattr*( ) then call *tcgetattr*( ) in order to determine what baud rates were actually selected.

RATIONALE

The *tcsetattr*( ) function can be interrupted in the following situations:

- It is interrupted while waiting for output to drain.
- It is called from a process in a background process group and SIGTTOU is caught.

See also the RATIONALE section in *tcgetattr*( ).

FUTURE DIRECTIONS

Using an input baud rate of 0 to set the input rate equal to the output rate may not necessarily be supported in a future version of this volume of IEEE Std 1003.1-200x.

SEE ALSO

cfgetispeed( ), *tcgetattr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<termios.h>**, **<unistd.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.
- The [EIO] error is added.

In the DESCRIPTION, the text describing use of *tcsetattr*( ) from a process which is a member of a background process group is clarified.

NAME

tcsetpgrp — set the foreground process group ID

SYNOPSIS

```
#include <unistd.h>

int tcsetpgrp(int fildes, pid_t pgid_id);
```

DESCRIPTION

If the process has a controlling terminal, *tcsetpgrp*( ) shall set the foreground process group ID associated with the terminal to *pgid_id*. The application shall ensure that the file associated with *fildes* is the controlling terminal of the calling process and the controlling terminal is currently associated with the session of the calling process. The application shall ensure that the value of *pgid_id* matches a process group ID of a process in the same session as the calling process.

Attempts to use *tcsetpgrp*( ) from a process which is a member of a background process group on a *fildes* associated with its controlling terminal shall cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process shall be allowed to perform the operation, and no signal is sent. |

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *tcsetpgrp*( ) function shall fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[EINVAL]	This implementation does not support the value in the *pgid_id* argument.
[ENOTTY]	The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
[EPERM]	The value of *pgid_id* is a value supported by the implementation, but does not match the process group ID of a process in the same session as the calling process.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

tcgetpgrp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 3.

Entry included for alignment with the POSIX.1-1988 standard.

Issue 6

In the SYNOPSIS, the inclusion of **<sys/types.h>** is no longer required.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- In the DESCRIPTION and ERRORS sections, text previously conditional on _POSIX_JOB_CONTROL is now mandated. This is a FIPS requirement.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The Open Group Corrigendum U047/4 is applied.

NAME

tdelete, tfind, tsearch, twalk — manage a binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void *tdelete(const void *restrict key, void **restrict rootp,
    int(*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
    int(*compar)(const void *, const void *));
void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
void twalk(const void *root,
    void (*action)(const void *, VISIT, int));
```

DESCRIPTION

The *tdelete*( ), *tfind*( ), *tsearch*( ), and *twalk*( ) functions manipulate binary search trees. Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The application shall ensure that the user-supplied routine returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The *tsearch*( ) function shall build and access the tree. The *key* argument is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by *key*, a pointer to this found node shall be returned. Otherwise, the value pointed to by *key* shall be inserted (that is, a new node is created and the value of *key* is copied to this node), and a pointer to this node returned. Only pointers are copied, so the application shall ensure that the calling routine stores the data. The *rootp* argument points to a variable that points to the root node of the tree. A null pointer value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable shall be set to point to the node which shall be at the root of the new tree.

Like *tsearch*( ), *tfind*( ) shall search for a node in the tree, returning a pointer to it if found. However, if it is not found, *tfind*( ) shall return a null pointer. The arguments for *tfind*( ) are the same as for *tsearch*( ).

The *tdelete*( ) function shall delete a node from a binary search tree. The arguments are the same as for *tsearch*( ). The variable pointed to by *rootp* shall be changed if the deleted node was the root of the tree. The *tdelete*( ) function shall return a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The *twalk*( ) function shall traverse a binary search tree. The *root* argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument shall be the address of the node being visited. The structure pointed to by this argument is unspecified and shall not be modified by the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-element to access the element stored in the node. The second argument shall be a value from an enumeration data type:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in **<search.h>**), depending on whether this is the first, second, or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

If the calling function alters the pointer to the root, the result is undefined.

RETURN VALUE

If the node is found, both *tsearch*( ) and *tfind*( ) shall return a pointer to it. If not, *tfind*( ) shall return a null pointer, and *tsearch*( ) shall return a pointer to the inserted item.

A null pointer shall be returned by *tsearch*( ) if there is not enough space available to create a new node.

A null pointer shall be returned by *tdelete*( ), *tfind*( ), and *tsearch*( ) if *rootp* is a null pointer on entry.

The *tdelete*( ) function shall return a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The *twalk*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <search.h>
#include <string.h>
#include <stdio.h>

#define STRSZ    10000
#define NODSZ    500

struct node {      /* Pointers to these are stored in the tree. */
    char    *string;
    int     length;
};

char   string_space[STRSZ];  /* Space to store strings. */
struct node nodes[NODSZ];    /* Nodes to store. */
void  *root = NULL;          /* This points to the root. */

int main(int argc, char *argv[])
{
    char   *strptr = string_space;
    struct node    *nodeptr = nodes;
    void   print_node(const void *, VISIT, int);
    int    i = 0, node_compare(const void *, const void *);

    while (gets(strptr) != NULL && i++ < NODSZ)  {
        /* Set node. */
        nodeptr->string = strptr;
        nodeptr->length = strlen(strptr);
        /* Put node into the tree. */
        (void) tsearch((void *)nodeptr, (void **)&root,
            node_compare);
```

```
            /* Adjust pointers, so we do not overwrite tree. */
            strptr += nodeptr->length + 1;
            nodeptr++;
        }
        twalk(root, print_node);
        return 0;
    }
    /*
     *  This routine compares two nodes, based on an
     *  alphabetical ordering of the string field.
     */
    int
    node_compare(const void *node1, const void *node2)
    {
        return strcmp(((const struct node *) node1)->string,
            ((const struct node *) node2)->string);
    }
    /*
     *  This routine prints out a node, the second time
     *  twalk encounters it or if it is a leaf.
     */
    void
    print_node(const void *ptr, VISIT order, int level)
    {
        const struct node *p = *(const struct node **) ptr;
        if (order == postorder || order == leaf)  {
            (void) printf("string = %s,  length = %d\n",
                p->string, p->length);
        }
    }
```

APPLICATION USAGE

The *root* argument to *twalk*( ) is one level of indirection less than the *rootp* arguments to *tdelete*( ) and *tsearch*( ).

There are two nomenclatures used to refer to the order in which tree nodes are visited. The *tsearch*( ) function uses **preorder**, **postorder**, and **endorder** to refer respectively to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternative nomenclature uses **preorder**, **inorder**, and **postorder** to refer to the same visits, which could result in some confusion over the meaning of **postorder**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate( ), *lsearch*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<search.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *tdelete*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

telldir — current location of a named directory stream

SYNOPSIS

XSI
```
#include <dirent.h>

long telldir(DIR *dirp);
```

DESCRIPTION

The *telldir*( ) function shall obtain the current location associated with the directory stream specified by *dirp*.

If the most recent operation on the directory stream was a *seekdir*( ), the directory position returned from the *telldir*( ) shall be the same as that supplied as a *loc* argument for *seekdir*( ).

RETURN VALUE

Upon successful completion, *telldir*( ) shall return the current location of the specified directory stream.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

opendir( ), *readdir*( ), *seekdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<dirent.h>**

CHANGE HISTORY

First released in Issue 2.

NAME

tempnam — create a name for a temporary file

SYNOPSIS

XSI

```
#include <stdio.h>

char *tempnam(const char *dir, const char *pfx);
```

DESCRIPTION

The *tempnam*( ) function shall generate a pathname that may be used for a temporary file.

The *tempnam*( ) function allows the user to control the choice of a directory. The *dir* argument points to the name of the directory in which the file is to be created. If *dir* is a null pointer or points to a string which is not a name for an appropriate directory, the path prefix defined as P_tmpdir in the **<stdio.h>** header shall be used. If that directory is not accessible, an implementation-defined directory may be used.

Many applications prefer their temporary files to have certain initial letter sequences in their names. The *pfx* argument should be used for this. This argument may be a null pointer or point to a string of up to five bytes to be used as the beginning of the filename.

Some implementations of *tempnam*( ) may use *tmpnam*( ) internally. On such implementations, if called more than {TMP_MAX} times in a single process, the behavior is implementation-defined.

RETURN VALUE

Upon successful completion, *tempnam*( ) shall allocate space for a string, put the generated pathname in that space, and return a pointer to it. The pointer shall be suitable for use in a subsequent call to *free*( ). Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *tempnam*( ) function shall fail if:

[ENOMEM] Insufficient storage space is available.

EXAMPLES

Generating a Pathname

The following example generates a pathname for a temporary file in directory **/tmp**, with the prefix *file*. After the filename has been created, the call to *free*( ) deallocates the space used to store the filename.

```
#include <stdio.h>
#include <stdlib.h>
...
char *directory = "/tmp";
char *fileprefix = "file";
char *file;

file = tempnam(directory, fileprefix);
free(file);
```

APPLICATION USAGE

This function only creates pathnames. It is the application's responsibility to create and remove the files. Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile*( ) more useful.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *free*( ), *open*( ), *tmpfile*( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The last paragraph of the DESCRIPTION was included as an APPLICATION USAGE note in previous issues.

NAME

tfind — search binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void *tfind(const void *key, void *const *rootp,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

Refer to *tdelete*( ).

NAME

tgamma, tgammaf, tgammal — compute gamma( ) function

SYNOPSIS

```
#include <math.h>

double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall compute the *gamma*( ) function of *x*.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return *Gamma*(*x*).

If *x* is a negative integer, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

If the correct value would cause overflow, a range error shall occur and *tgamma*( ), *tgammaf*( ), and *tgammal*( ) shall return the value of the macro HUGE_VAL, HUGE_VALF, or HUGE_VALL, respectively.

MX If *x* is NaN, a NaN shall be returned.

If *x* is +Inf, *x* shall be returned.

If *x* is ±0, a pole error shall occur, and *tgamma*( ), *tgammaf*( ), and *tgammal*( ) shall return ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL, respectively.

If *x* is –Inf, a domain error shall occur, and either a NaN (if supported), or an implementation-defined value shall be returned.

ERRORS

These functions shall fail if:

MX Domain Error — The value of *x* is a negative integer, or *x* is –Inf.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

MX Pole Error — The value of *x* is zero.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the divide-by-zero floating-point exception shall be raised.

Range Error — The value overflows.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised. |

EXAMPLES

None.

APPLICATION USAGE

For IEEE Std 754-1985 **double**, overflow happens when $0 < x < 1/\text{DBL\_MAX}$, and $171.7 < x$. Overflow also happens near negative integers.

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

This function is named *tgamma*( ) in order to avoid conflicts with the historical *gamma*( ) and *lgamma*( ) functions.

FUTURE DIRECTIONS

It is possible that the error response for a negative integer argument may be changed to a pole error and a return value of ±Inf.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *lgamma*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>** |

CHANGE HISTORY

First released in Issue 6. Derived from the ISO/IEC 9899:1999 standard.

NAME

time — get time

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

CX The *time*( ) function shall return the value of time in seconds since the Epoch.

The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer, no value is stored.

RETURN VALUE

Upon successful completion, *time*( ) shall return the value of time. Otherwise, (**time_t**)–1 shall be returned.

ERRORS

No errors are defined.

EXAMPLES

Getting the Current Time

The following example uses the *time*( ) function to calculate the time elapsed, in seconds, since January 1, 1970 0:00 UTC, *localtime*( ) to convert that value to a broken-down time, and *asctime*( ) to convert the broken-down time values into a printable string.

```
#include <stdio.h>
#include <time.h>

main()
{
time_t result;

    result = time(NULL);
    printf("%s%ld secs since the Epoch\n",
        asctime(localtime(&result)),
            (long)result);
    return(0);
}
```

This example writes the current time to *stdout* in a form like this:

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

Timing an Event

The following example gets the current time, prints it out in the user's format, and prints the number of minutes to an event being timed.

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
    minutes_to_event);
...
```

APPLICATION USAGE

None.

RATIONALE

The *time*( ) function returns a value in seconds (type **time_t**) while *times*( ) returns a set of values in clock ticks (type **clock_t**). Some historical implementations, such as 4.3 BSD, have mechanisms capable of returning more precise times (see below). A generalized timing scheme to unify these various timing mechanisms has been proposed but not adopted.

Implementations in which **time_t** is a 32-bit signed integer (many historical implementations) fail in the year 2038. IEEE Std 1003.1-200x does not address this problem. However, the use of the **time_t** type is mandated in order to ease the eventual fix.

The use of the **<time.h>**, header instead of **<sys/types.h>**, allows compatibility with the ISO C standard.

Many historical implementations (including Version 7) and the 1984 /usr/group standard use **long** instead of **time_t**. This volume of IEEE Std 1003.1-200x uses the latter type in order to agree with the ISO C standard.

4.3 BSD includes *time*( ) only as an alternate function to the more flexible *gettimeofday*( ) function.

FUTURE DIRECTIONS

In a future version of this volume of IEEE Std 1003.1-200x, **time_t** is likely to be required to be capable of representing times far in the future. Whether this will be mandated as a 64-bit type or a requirement that a specific date in the future be representable (for example, 10000 AD) is not yet determined. Systems purchased after the approval of this volume of IEEE Std 1003.1-200x should be evaluated to determine whether their lifetime will extend past 2038.

SEE ALSO

asctime( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *mktime*( ), *strftime*( ), *strptime*( ), *utime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked. |

The EXAMPLES, RATIONALE, and FUTURE DIRECTIONS sections are added. |

NAME

timer_create — create a per-process timer (**REALTIME**)

SYNOPSIS

TMR

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid, struct sigevent *restrict evp,
    timer_t *restrict timerid);
```

DESCRIPTION

The *timer_create*( ) function shall create a per-process timer using the specified clock, *clock_id*, as the timing base. The *timer_create*( ) function shall return, in the location referenced by *timerid*, a timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID shall be unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is defined in **<time.h>**. The timer whose ID is returned shall be in a disarmed state upon return from *timer_create*( ).

The *evp* argument, if non-NULL, points to a **sigevent** structure. This structure, allocated by the application, defines the asynchronous notification to occur as specified in Section 2.4.1 (on page 478) when the timer expires. If the *evp* argument is NULL, the effect is as if the *evp* argument pointed to a **sigevent** structure with the *sigev_notify* member having the value SIGEV_SIGNAL, the *sigev_signo* having a default signal number, and the *sigev_value* member having the value of the timer ID.

Each implementation shall define a set of clocks that can be used as timing bases for per-process timers. All implementations shall support a *clock_id* of CLOCK_REALTIME. MON If the Monotonic Clock option is supported, implementations shall support a *clock_id* of CLOCK_MONOTONIC.

Per-process timers shall not be inherited by a child process across a *fork*( ) and shall be disarmed and deleted by an *exec*.

CPT If _POSIX_CPUTIME is defined, implementations shall support *clock_id* values representing the CPU-time clock of the calling process.

TCT If _POSIX_THREAD_CPUTIME is defined, implementations shall support *clock_id* values representing the CPU-time clock of the calling thread.

CPT|TCT It is implementation-defined whether a *timer_create*( ) function will succeed if the value defined by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process or thread invoking the function.

RETURN VALUE

If the call succeeds, *timer_create*( ) shall return zero and update the location referenced by *timerid* to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, the function shall return a value of –1 and set *errno* to indicate the error. The value of *timerid* is undefined if an error occurs.

ERRORS

The *timer_create*( ) function shall fail if:

[EAGAIN]	The system lacks sufficient signal queuing resources to honor the request.
[EAGAIN]	The calling process has already created all of the timers it is allowed by this implementation.
[EINVAL]	The specified clock ID is not defined.

CPT|TCT [ENOTSUP] The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by *clock_id* and associated with a process or thread different from the process or thread invoking *timer_create*( ).

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Periodic Timer Overrun and Resource Allocation

The specified timer facilities may deliver realtime signals (that is, queued signals) on implementations that support this option. Since realtime applications cannot afford to lose notifications of asynchronous events, like timer expirations or asynchronous I/O completions, it must be possible to ensure that sufficient resources exist to deliver the signal when the event occurs. In general, this is not a difficulty because there is a one-to-one correspondence between a request and a subsequent signal generation. If the request cannot allocate the signal delivery resources, it can fail the call with an [EAGAIN] error.

Periodic timers are a special case. A single request can generate an unspecified number of signals. This is not a problem if the requesting process can service the signals as fast as they are generated, thus making the signal delivery resources available for delivery of subsequent periodic timer expiration signals. But, in general, this cannot be assured—processing of periodic timer signals may ‘‘overrun’’; that is, subsequent periodic timer expirations may occur before the currently pending signal has been delivered.

Also, for signals, according to the POSIX.1-1990 standard, if subsequent occurrences of a pending signal are generated, it is implementation-defined whether a signal is delivered for each occurrence. This is not adequate for some realtime applications. So a mechanism is required to allow applications to detect how many timer expirations were delayed without requiring an indefinite amount of system resources to store the delayed expirations.

The specified facilities provide for an overrun count. The overrun count is defined as the number of extra timer expirations that occurred between the time a timer expiration signal is generated and the time the signal is delivered. The signal-catching function, if it is concerned with overruns, can retrieve this count on entry. With this method, a periodic timer only needs one ‘‘signal queuing resource’’ that can be allocated at the time of the *timer_create*( ) function call.

A function is defined to retrieve the overrun count so that an application need not allocate static storage to contain the count, and an implementation need not update this storage asynchronously on timer expirations. But, for some high-frequency periodic applications, the overhead of an additional system call on each timer expiration may be prohibitive. The functions, as defined, permit an implementation to maintain the overrun count in user space, associated with the *timerid*. The *timer_getoverrun*( ) function can then be implemented as a macro that uses the *timerid* argument (which may just be a pointer to a user space structure containing the counter) to locate the overrun count with no system call overhead. Other implementations, less concerned with this class of applications, can avoid the asynchronous update of user space by maintaining the count in a system structure at the cost of the extra system call to obtain it.

Timer Expiration Signal Parameters

The Realtime Signals Extension option supports an application-specific datum that is delivered to the extended signal handler. This value is explicitly specified by the application, along with the signal number to be delivered, in a **sigevent** structure. The type of the application-defined value can be either an integer constant or a pointer. This explicit specification of the value, as opposed to always sending the timer ID, was selected based on existing practice.

It is common practice for realtime applications (on non-POSIX systems or realtime extended POSIX systems) to use the parameters of event handlers as the case label of a switch statement or as a pointer to an application-defined data structure. Since *timer_id*s are dynamically allocated by the *timer_create*( ) function, they can be used for neither of these functions without additional application overhead in the signal handler; for example, to search an array of saved timer IDs to associate the ID with a constant or application data structure. |

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *timer_delete*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *timer_create*( ) function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

CPU-time clocks are added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding the requirement for the CLOCK_MONOTONIC clock under the Monotonic Clock option.

The **restrict** keyword is added to the *timer_create*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

timer_delete — delete a per-process timer (**REALTIME**)

SYNOPSIS

TMR
```
#include <time.h>

int timer_delete(timer_t timerid);
```

DESCRIPTION

The *timer_delete*( ) function deletes the specified timer, *timerid*, previously created by the *timer_create*( ) function. If the timer is armed when *timer_delete*( ) is called, the behavior shall be as if the timer is automatically disarmed before removal. The disposition of pending signals for the deleted timer is unspecified.

RETURN VALUE

If successful, the *timer_delete*( ) function shall return a value of zero. Otherwise, the function shall return a value of −1 and set *errno* to indicate the error.

ERRORS

The *timer_delete*( ) function shall fail if:

[EINVAL]	The timer ID specified by *timerid* is not a valid timer ID.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

timer_create( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *timer_delete*( ) function is marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

NAME

timer_getoverrun, timer_gettime, timer_settime — per-process timers (**REALTIME**)

SYNOPSIS

TMR

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
    const struct itimerspec *restrict value,
    struct itimerspec *restrict ovalue);
```

DESCRIPTION

The *timer_gettime*( ) function shall store the amount of time until the specified timer, *timerid*, expires and the reload value of the timer into the space pointed to by the *value* argument. The *it_value* member of this structure shall contain the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until timer expiration, even if the timer was armed with absolute time. The *it_interval* member of *value* shall contain the reload value last set by *timer_settime*( ).

The *timer_settime*( ) function shall set the time until the next expiration of the timer specified by *timerid* from the *it_value* member of the *value* argument and arms the timer if the *it_value* member of *value* is non-zero. If the specified timer was already armed when *timer_settime*( ) is called, this call shall reset the time until next expiration to the *value* specified. If the *it_value* member of *value* is zero, the timer shall be disarmed. The effect of disarming or resetting a timer with pending expiration notifications is unspecified.

If the flag TIMER_ABSTIME is not set in the argument *flags*, *timer_settime*( ) shall behave as if the time until next expiration is set to be equal to the interval specified by the *it_value* member of *value*. That is, the timer shall expire in *it_value* nanoseconds from when the call is made. If the flag TIMER_ABSTIME is set in the argument *flags*, *timer_settime*( ) shall behave as if the time until next expiration is set to be equal to the difference between the absolute time specified by the *it_value* member of *value* and the current value of the clock associated with *timerid*. That is, the timer shall expire when the clock reaches the value specified by the *it_value* member of *value*. If the specified time has already passed, the function shall succeed and the expiration notification shall be made.

The reload value of the timer shall be set to the value specified by the *it_interval* member of *value*. When a timer is armed with a non-zero *it_interval*, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer shall be rounded up to the larger multiple of the resolution. Quantization error shall not cause the timer to expire earlier than the rounded time value.

If the argument *ovalue* is not NULL, the function *timer_settime*( ) shall store, in the location referenced by *ovalue*, a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers shall not expire before their scheduled time.

Only a single signal shall be queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal shall be queued, and a timer overrun shall occur. RTS When a timer expiration signal is delivered to or accepted by a process, if the implementation supports the Realtime Signals Extension, the *timer_getoverrun*( ) function shall return the timer expiration overrun count for the specified timer. The overrun count returned contains the number of extra timer expirations that occurred between the time the signal was generated (queued) and when it was delivered or accepted, up to but not including an implementation-defined maximum of {DELAYTIMER_MAX}. If the number of such extra expirations is greater than or equal to {DELAYTIMER_MAX}, then the overrun count shall be set to {DELAYTIMER_MAX}. The value returned by *timer_getoverrun*( ) shall apply to the most recent expiration signal delivery or acceptance for the timer. If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of *timer_getoverrun*( ) is unspecified.

RETURN VALUE

If the *timer_getoverrun*( ) function succeeds, it shall return the timer expiration overrun count as explained above.

If the *timer_gettime*( ) or *timer_settime*( ) functions succeed, a value of 0 shall be returned.

If an error occurs for any of these functions, the value −1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions shall if:

[EINVAL] The *timerid* argument does not correspond to an ID returned by *timer_create*( ) but not yet deleted by *timer_delete*( ).

The *timer_settime*( ) function shall fail if:

[EINVAL] A *value* structure specified a nanosecond value less than zero or greater than or equal to 1,000 million, and the *it_value* member of that structure did not specify zero seconds and nanoseconds.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Practical clocks tick at a finite rate, with rates of 100 Hertz and 1,000 Hertz being common. The inverse of this tick rate is the clock resolution, also called the clock granularity, which in either case is expressed as a time duration, being 10 milliseconds and 1 millisecond respectively for these common rates. The granularity of practical clocks implies that if one reads a given clock twice in rapid succession, one may get the same time value twice; and that timers must wait for the next clock tick after the theoretical expiration time, to ensure that a timer never returns too soon. Note also that the granularity of the clock may be significantly coarser than the resolution of the data format used to set and get time and interval values. Also note that some implementations may choose to adjust time and/or interval values to exactly match the ticks of the underlying clock.

This volume of IEEE Std 1003.1-200x defines functions that allow an application to determine the implementation-supported resolution for the clocks and requires an implementation to document the resolution supported for timers and *nanosleep*( ) if they differ from the supported clock resolution. This is more of a procurement issue than a runtime application issue.

FUTURE DIRECTIONS

None.

SEE ALSO

clock_getres( ), *timer_create*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with the POSIX Realtime Extension.

Issue 6

The *timer_getoverrun*( ), *timer_gettime*( ), and *timer_settime*( ) functions are marked as part of the Timers option.

The [ENOSYS] error condition has been removed as stubs need not be provided if an implementation does not support the Timers option.

The [EINVAL] error condition is updated to include the following: ‘‘and the *it_value* member of that structure did not specify zero seconds and nanoseconds.’’ This change is for IEEE PASC Interpretation 1003.1 #89.

The DESCRIPTION for *timer_getoverrun*( ) is updated to clarify that ‘‘If no expiration signal has been delivered for the timer, or if the Realtime Signals Extension is not supported, the return value of *timer_getoverrun*( ) is unspecified’’.

The **restrict** keyword is added to the *timer_settime*( ) prototype for alignment with the ISO / IEC 9899: 1999 standard.

NAME

times — get process and waited-for child process times

SYNOPSIS

```
#include <sys/times.h>

clock_t times(struct tms *buffer);
```

DESCRIPTION

The *times*( ) function shall fill the **tms** structure pointed to by *buffer* with time-accounting information. The **tms** structure is defined in **<sys/times.h>**.

All times are measured in terms of the number of clock ticks used.

The times of a terminated child process shall be included in the *tms_cutime* and *tms_cstime* elements of the parent when *wait*( ) or *waitpid*( ) returns the process ID of this terminated child. If a child process has not waited for its children, their times shall not be included in its times.

- The *tms_utime* structure member is the CPU time charged for the execution of user instructions of the calling process.
- The *tms_stime* structure member is the CPU time charged for execution by the system on behalf of the calling process.
- The *tms_cutime* structure member is the sum of the *tms_utime* and *tms_cutime* times of the child processes.
- The *tms_cstime* structure member is the sum of the *tms_stime* and *tms_cstime* times of the child processes.

RETURN VALUE

Upon successful completion, *times*( ) shall return the elapsed real time, in clock ticks, since an arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of *times*( ) within the process to another. The return value may overflow the possible range of type **clock_t**. If *times*( ) fails, (**clock_t**)−1 shall be returned and *errno* set to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

Timing a Database Lookup

The following example defines two functions, *start_clock*( ) and *end_clock*( ), that are used to time a lookup. It also defines variables of type **clock_t** and **tms** to measure the duration of transactions. The *start_clock*( ) function saves the beginning times given by the *times*( ) function. The *end_clock*( ) function gets the ending times and prints the difference between the two times.

```
#include <sys/times.h>
#include <stdio.h>
...
void start_clock(void);
void end_clock(char *msg);
...
static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;
```

```
...
void
start_clock()
{
    st_time = times(&st_cpu);
}

void
end_clock(char *msg)
{
    en_time = times(&en_cpu);

    printf(msg);
    printf("Real Time: %ld, User Time %ld, System Time %ld\n",
        en_time - st_time,
        en_cpu.tms_utime - st_cpu.tms_utime,
        en_cpu.tms_stime - st_cpu.tms_stime);
}
```

APPLICATION USAGE

Applications should use *sysconf*(_SC_CLK_TCK) to determine the number of clock ticks per second as it may vary from system to system.

RATIONALE

The accuracy of the times reported is intentionally left unspecified to allow implementations flexibility in design, from uniprocessor to multi-processor networks.

The inclusion of times of child processes is recursive, so that a parent process may collect the total times of all of its descendants. But the times of a child are only added to those of its parent when its parent successfully waits on the child. Thus, it is not guaranteed that a parent process can always see the total times of all its descendants; see also the discussion of the term *realtime* in *alarm*( ).

If the type **clock_t** is defined to be a signed 32-bit integer, it overflows in somewhat more than a year if there are 60 clock ticks per second, or less than a year if there are 100. There are individual systems that run continuously for longer than that. This volume of IEEE Std 1003.1-200x permits an implementation to make the reference point for the returned value be the start-up time of the process, rather than system start-up time.

The term *charge* in this context has nothing to do with billing for services. The operating system accounts for time used in this way. That information must be correct, regardless of how that information is used.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *fork*( ), *sysconf*( ), *time*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/times.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

timezone — difference from UTC and local standard time

SYNOPSIS

XSI

```
#include <time.h>

extern long timezone;
```

DESCRIPTION

Refer to *tzset*( ).

NAME

tmpfile — create a temporary file

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *tmpfile*( ) function shall create a temporary file and open a corresponding stream. The file shall be automatically deleted when all references to the file are closed. The file is opened as in *fopen*( ) for update (*w*+).

CX In some implementations, a permanent file may be left behind if the process calling *tmpfile*( ) is killed while it is processing a call to *tmpfile*( ).

An error message may be written to standard error if the stream cannot be opened.

RETURN VALUE

Upon successful completion, *tmpfile*( ) shall return a pointer to the stream of the file that is created. Otherwise, it shall return a null pointer CX and set *errno* to indicate the error.

ERRORS

The *tmpfile*( ) function shall fail if:

CX	[EINTR]	A signal was caught during *tmpfile*( ).
CX	[EMFILE]	{OPEN_MAX} file descriptors are currently open in the calling process.
CX	[ENFILE]	The maximum allowable number of files is currently open in the system.
CX	[ENOSPC]	The directory or file system which would contain the new file cannot be expanded.
CX	[EOVERFLOW]	The file is a regular file and the size of the file cannot be represented correctly in an object of type **off_t**.

The *tmpfile*( ) function may fail if:

CX	[EMFILE]	{FOPEN_MAX} streams are currently open in the calling process.
CX	[ENOMEM]	Insufficient storage space is available.

EXAMPLES

Creating a Temporary File

The following example creates a temporary file for update, and returns a pointer to a stream for the created file in the *fp* variable.

```
#include <stdio.h>
...
FILE *fp;

fp = tmpfile ();
```

APPLICATION USAGE

It should be possible to open at least {TMP_MAX} temporary files during the lifetime of the program (this limit may be shared with *tmpnam*( )) and there should be no limit on the number simultaneously open other than this limit and any limit on the number of open files ({FOPEN_MAX}).

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *tmpnam*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

Large File Summit extensions are added.

The last two paragraphs of the DESCRIPTION were included as APPLICATION USAGE notes in previous issues.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the ERRORS section, the [EOVERFLOW] condition is added. This change is to support large files.
- The [EMFILE] optional error condition is added.

The APPLICATION USAGE section is added for alignment with the ISO/IEC 9899:1999 standard.

NAME

tmpnam — create a name for a temporary file

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *s);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *tmpnam*( ) function shall generate a string that is a valid filename and that is not the same as the name of an existing file. The function is potentially capable of generating {TMP_MAX} different strings, but any or all of them may already be in use by existing files and thus not be suitable return values.

The *tmpnam*( ) function generates a different string each time it is called from the same process, up to {TMP_MAX} times. If it is called more than {TMP_MAX} times, the behavior is implementation-defined.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *tmpnam*( ).

CX If the application uses any of the functions guaranteed to be available if either _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS is defined, the application shall ensure that the *tmpnam*( ) function is called with a non-NULL parameter.

RETURN VALUE

Upon successful completion, *tmpnam*( ) shall return a pointer to a string. If no suitable string can be generated, the *tmpnam*( ) function shall return a null pointer.

If the argument *s* is a null pointer, *tmpnam*( ) shall leave its result in an internal static object and return a pointer to that object. Subsequent calls to *tmpnam*( ) may modify the same object. If the argument *s* is not a null pointer, it is presumed to point to an array of at least L_tmpnam **char**s; *tmpnam*( ) shall write its result in that array and shall return the argument as its value.

ERRORS

No errors are defined.

EXAMPLES

Generating a Filename

The following example generates a unique filename and stores it in the array pointed to by *ptr*.

```
#include <stdio.h>
...
char filename[L_tmpnam+1];
char *ptr;

ptr = tmpnam(filename);
```

APPLICATION USAGE

This function only creates filenames. It is the application's responsibility to create and remove the files.

Between the time a pathname is created and the file is opened, it is possible for some other process to create a file with the same name. Applications may find *tmpfile*( ) more useful. |

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fopen( ), *open*( ), *tempnam*( ), *tmpfile*( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The DESCRIPTION is expanded for alignment with the ISO/IEC 9899:1999 standard.

NAME

toascii — translate integer to a 7-bit ASCII character

SYNOPSIS

XSI

```
#include <ctype.h>

int toascii(int c);
```

DESCRIPTION

The *toascii*( ) function shall convert its argument into a 7-bit ASCII character.

RETURN VALUE

The *toascii*( ) function shall return the value (*c* &0x7f).

ERRORS

No errors are returned.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

isascii( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

tolower — transliterate uppercase characters to lowercase

SYNOPSIS

```
#include <ctype.h>

int tolower(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *tolower*( ) function has as a domain a type **int**, the value of which is representable as an **unsigned char** or the value of EOF. If the argument has any other value, the behavior is undefined. If the argument of *tolower*( ) represents an uppercase letter, and there exists a corresponding lowercase letter CX (as defined by character type information in the program locale category *LC_CTYPE*), the result shall be the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

RETURN VALUE

Upon successful completion, *tolower*( ) shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

toupper — transliterate lowercase characters to uppercase

SYNOPSIS

```
#include <ctype.h>

int toupper(int c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *toupper*( ) function has as a domain a type **int**, the value of which is representable as an **unsigned char** or the value of EOF. If the argument has any other value, the behavior is undefined. If the argument of *toupper*( ) represents a lowercase letter, and there exists a corresponding uppercase letter (as defined by character type information in the program locale category *LC_CTYPE*), the result shall be the corresponding uppercase letter. All other arguments in the domain are returned unchanged.

RETURN VALUE

Upon successful completion, *toupper*( ) shall return the uppercase letter corresponding to the argument passed.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ctype.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

towctrans — wide-character transliteration

SYNOPSIS

```
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *towctrans*( ) function shall transliterate the wide-character code *wc* using the mapping described by *desc*. The current setting of the *LC_CTYPE* category should be the same as during the call to *wctrans*( ) that returned the value *desc*. CX If the value of *desc* is invalid (that is, not obtained by a call to *wctrans*( ) or *desc* is invalidated by a subsequent call to *setlocale*( ) that has affected category *LC_CTYPE*), the result is unspecified.

An application wishing to check for error situations should set *errno* to 0 before calling *towctrans*( ). If *errno* is non-zero on return, an error has occurred.

RETURN VALUE

If successful, the *towctrans*( ) function shall return the mapped value of *wc* using the mapping described by *desc*. Otherwise, it shall return *wc* unchanged.

ERRORS

The *towctrans*( ) function may fail if:

CX [EINVAL] *desc* contains an invalid transliteration descriptor.

EXAMPLES

None.

APPLICATION USAGE

The strings `"tolower"` and `"toupper"` are reserved for the standard mapping names. In the table below, the functions in the left column are equivalent to the functions in the right column.

```
towlower(wc)    towctrans(wc, wctrans("tolower"))
towupper(wc)    towctrans(wc, wctrans("toupper"))
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

towlower( ), *towupper*( ), *wctrans*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**

CHANGE HISTORY

First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

Extensions beyond the ISO C standard are now marked.

NAME

towlower — transliterate uppercase wide-character code to lowercase

SYNOPSIS

```
#include <wctype.h>

wint_t towlower(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *towlower*( ) function has as a domain a type **wint_t**, the value of which the application shall ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a valid character in the current locale or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of *towlower*( ) represents an uppercase wide-character code, and there exists a corresponding lowercase wide-character code (as defined by character type information in the program locale category *LC_CTYPE*), the result shall be the corresponding lowercase wide-character code. All other arguments in the domain are returned unchanged.

RETURN VALUE

Upon successful completion, *towlower*( ) shall return the lowercase letter corresponding to the argument passed; otherwise, it shall return the argument unchanged.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

towupper — transliterate lowercase wide-character code to uppercase

SYNOPSIS

```
#include <wctype.h>

wint_t towupper(wint_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *towupper*( ) function has as a domain a type **wint_t**, the value of which the application shall ensure is a character representable as a **wchar_t**, and a wide-character code corresponding to a valid character in the current locale or the value of WEOF. If the argument has any other value, the behavior is undefined. If the argument of *towupper*( ) represents a lowercase wide-character code, and there exists a corresponding uppercase wide-character code (as defined by character type information in the program locale category *LC_CTYPE*), the result shall be the corresponding uppercase wide-character code. All other arguments in the domain are returned unchanged.

RETURN VALUE

Upon successful completion, *towupper*( ) shall return the uppercase letter corresponding to the argument passed. Otherwise, it shall return the argument unchanged.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

setlocale( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

trunc, truncf, truncl — round to truncated integer value

SYNOPSIS

```
#include <math.h>

double trunc(double x);
float truncf(float x);
long double truncl(long double x);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

RETURN VALUE

Upon successful completion, these functions shall return the truncated integer value.

MX If *x* is NaN, a NaN shall be returned.

If *x* is ±0, or ±Inf, *x* shall be returned.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<math.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

truncate — truncate a file to a specified length

SYNOPSIS

XSI

```
#include <unistd.h>

int truncate(const char *path, off_t length);
```

DESCRIPTION

The *truncate*( ) function shall cause the regular file named by *path* to have a size which shall be equal to *length* bytes.

If the file previously was larger than *length*, the extra data is discarded. If the file was previously shorter than *length*, its size is increased, and the extended area appears as if it were zero-filled.

The application shall ensure that the process has write permission for the file.

If the request would cause the file size to exceed the soft file size limit for the process, the request shall fail and the implementation shall generate the SIGXFSZ signal for the process.

This function shall not modify the file offset for any open file descriptions associated with the file. Upon successful completion, if the file size is changed, this function shall mark for update the *st_ctime* and *st_mtime* fields of the file, and the S_ISUID and S_ISGID bits of the file mode may be cleared.

RETURN VALUE

Upon successful completion, *truncate*( ) shall return 0. Otherwise, –1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *truncate*( ) function shall fail if:

[EINTR]	A signal was caught during execution.
[EINVAL]	The *length* argument was less than 0.
[EFBIG] or [EINVAL]	The *length* argument was greater than the maximum file size.
[EIO]	An I/O error occurred while reading from or writing to a file system.
[EACCES]	A component of the path prefix denies search permission, or write permission is denied on the file.
[EISDIR]	The named file is a directory.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix of *path* is not a directory.
[EROFS]	The named file resides on a read-only file system.

The *truncate*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X / OPEN UNIX extension to BASE.

Large File Summit extensions are added.

Issue 6

This reference page is split out from the *ftruncate*( ) reference page.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

truncf, truncl — round to truncated integer value

SYNOPSIS

```
#include <math.h>

float truncf(float x);
long double truncl(long double x);
```

DESCRIPTION

Refer to *trunc*( ).

NAME

tsearch — search a binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
```

DESCRIPTION

Refer to *tdelete*( ).

NAME

ttyname, ttyname_r — find pathname of a terminal

SYNOPSIS

```
#include <unistd.h>

char *ttyname(int fildes);
TSF int ttyname_r(int fildes, char *name, size_t namesize);
```

DESCRIPTION

The *ttyname*( ) function shall return a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *fildes*. The return value may point to static data whose content is overwritten by each call.

The *ttyname*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

TSF The *ttyname_r*( ) function shall store the null-terminated pathname of the terminal associated with the file descriptor *fildes* in the character array referenced by *name*. The array is *namesize* characters long and should have space for the name and the terminating null character. The maximum length of the terminal name shall be {TTY_NAME_MAX}.

RETURN VALUE

Upon successful completion, *ttyname*( ) shall return a pointer to a string. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

TSF If successful, the *ttyname_r*( ) function shall return zero. Otherwise, an error number shall be returned to indicate the error.

ERRORS

The *ttyname*( ) function may fail if:

[EBADF]	The *fildes* argument is not a valid file descriptor.
[ENOTTY]	The *fildes* argument does not refer to a terminal.

The *ttyname_r*( ) function may fail if:

TSF	[EBADF]	The *fildes* argument is not a valid file descriptor.
TSF	[ENOTTY]	The *fildes* argument does not refer to a terminal.
TSF	[ERANGE]	The value of *namesize* is smaller than the length of the string to be returned including the terminating null character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The term *terminal* is used instead of the historical term *terminal device* in order to avoid a reference to an undefined term.

The thread-safe version places the terminal name in a user-supplied buffer and returns a non-zero value if it fails. The non-thread-safe version may return the name in a static data area that may be overwritten by each call.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The *ttyname_r*( ) function is included for alignment with the POSIX Threads Extension.

A note indicating that the *ttyname*( ) function need not be reentrant is added to the DESCRIPTION.

Issue 6

The *ttyname_r*( ) function is marked as part of the Thread-Safe Functions option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The statement that *errno* is set on error is added.
- The [EBADF] and [ENOTTY] optional error conditions are added.

NAME

twalk — traverse a binary search tree

SYNOPSIS

XSI

```
#include <search.h>

void twalk(const void *root,
    void (*action)(const void *, VISIT, int ));
```

DESCRIPTION

Refer to *tdelete*( ).

NAME

tzname — timezone strings

SYNOPSIS

CX
```
#include <time.h>

extern char *tzname[2];
```

DESCRIPTION

Refer to *tzset*( ).

NAME

daylight, timezone, tzname, tzset — set timezone conversion information

SYNOPSIS

```
#include <time.h>
```

XSI

```
extern int daylight;
extern long timezone;
```

CX

```
extern char *tzname[2];
void tzset(void);
```

DESCRIPTION

The *tzset*( ) function shall use the value of the environment variable *TZ* to set time conversion information used by *ctime*( ), *localtime*( ), *mktime*( ), and *strftime*( ). If *TZ* is absent from the environment, implementation-defined default timezone information shall be used.

The *tzset*( ) function shall set the external variable *tzname* as follows:

```
tzname[0] = "std";
tzname[1] = "dst";
```

where *std* and *dst* are as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

XSI The *tzset*( ) function also shall set the external variable *daylight* to 0 if Daylight Savings Time conversions should never be applied for the timezone in use; otherwise, non-zero. The external variable *timezone* shall be set to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time.

RETURN VALUE

The *tzset*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

Example TZ variables and their timezone differences are given in the table below:

TZ	*timezone*
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	–9*60*60
MET-1MEST	–1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ctime( ), *localtime*( ), *mktime*( ), *strftime*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<time.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

The example is corrected.

NAME

ualarm — set the interval timer

SYNOPSIS

OB XSI

```
#include <unistd.h>

useconds_t ualarm(useconds_t useconds, useconds_t interval);
```

DESCRIPTION

The *ualarm*( ) function shall cause the SIGALRM signal to be generated for the calling process after the number of realtime microseconds specified by the *useconds* argument has elapsed. When the *interval* argument is non-zero, repeated timeout notification occurs with a period in microseconds specified by the *interval* argument. If the notification signal, SIGALRM, is not caught or ignored, the calling process is terminated.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value.

Interactions between *ualarm*( ) and any of the following are unspecified:

alarm( )
nanosleep( )
setitimer( )
timer_create( )
timer_delete( )
timer_getoverrun( )
timer_gettime( )
timer_settime( )
sleep( )

RETURN VALUE

The *ualarm*( ) function shall return the number of microseconds remaining from the previous *ualarm*( ) call. If no timeouts are pending or if *ualarm*( ) has not previously been called, *ualarm*( ) shall return 0.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

Applications are recommended to use *nanosleep*( ) if the Timers option is supported, or *setitimer*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), or *timer_settime*( ) instead of this function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *nanosleep*( ), *setitimer*( ), *sleep*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked obsolescent.

NAME

ulimit — get and set process limits

SYNOPSIS

XSI
```
#include <ulimit.h>

long ulimit(int cmd, ...);
```

DESCRIPTION

The *ulimit*( ) function shall control process limits. The process limits that can be controlled by this function include the maximum size of a single file that can be written (this is equivalent to using *setrlimit*( ) with RLIMIT_FSIZE). The *cmd* values, defined in **<ulimit.h>** include: |

UL_GETFSIZE	Return the file size limit (RLIMIT_FSIZE) of the process. The limit shall be in units of 512-byte blocks and shall be inherited by child processes. Files of any size can be read. The return value shall be the integer part of the soft file size limit divided by 512. If the result cannot be represented as a **long**, the result is unspecified. |
UL_SETFSIZE	Set the file size limit for output operations of the process to the value of the second argument, taken as a **long**, multiplied by 512. If the result would overflow an **rlim_t**, the actual value set is unspecified. Any process may decrease its own limit, but only a process with appropriate privileges may increase the limit. The return value shall be the integer part of the new file size limit divided by 512.

The *ulimit*( ) function shall not change the setting of *errno* if successful.

As all return values are permissible in a successful situation, an application wishing to check for error situations should set *errno* to 0, then call *ulimit*( ), and, if it returns –1, check to see if *errno* is non-zero.

RETURN VALUE

Upon successful completion, *ulimit*( ) shall return the value of the requested limit. Otherwise, –1 shall be returned and *errno* set to indicate the error.

ERRORS

The *ulimit*( ) function shall fail and the limit shall be unchanged if:

[EINVAL]	The *cmd* argument is not valid.
[EPERM]	A process not having appropriate privileges attempts to increase its file size limit.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

getrlimit( ), *setrlimit*( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<ulimit.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

In the description of UL_SETFSIZE, the text is corrected to refer to **rlim_t** rather than the spurious **rlimit_t**.

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

NAME

umask — set and get file mode creation mask

SYNOPSIS

```
#include <sys/stat.h>

mode_t umask(mode_t cmask);
```

DESCRIPTION

The *umask*( ) function shall set the process' file mode creation mask to *cmask* and return the previous value of the mask. Only the file permission bits of *cmask* (see **<sys/stat.h>**) are used; the meaning of the other bits is implementation-defined.

The process' file mode creation mask is used during *open*( ), *creat*( ), *mkdir*( ), and *mkfifo*( ) to turn off permission bits in the *mode* argument supplied. Bit positions that are set in *cmask* are cleared in the mode of the created file.

RETURN VALUE

The file permission bits in the value returned by *umask*( ) shall be the previous value of the file mode creation mask. The state of any other bits in that value is unspecified, except that a subsequent call to *umask*( ) with the returned value as *cmask* shall leave the state of the mask the same as its state before the first call, including any unspecified use of those bits.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Unsigned argument and return types for *umask*( ) were proposed. The return type and the argument were both changed to **mode_t**.

Historical implementations have made use of additional bits in *cmask* for their implementation-defined purposes. The addition of the text that the meaning of other bits of the field is implementation-defined permits these implementations to conform to this volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

creat( ), *mkdir*( ), *mkfifo*( ), *open*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/stat.h>**, **<sys/types.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed. |

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

NAME

uname — get name of current system

SYNOPSIS

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

DESCRIPTION

The *uname*( ) function shall store information identifying the current system in the structure pointed to by *name*.

The *uname*( ) function uses the **utsname** structure defined in **<sys/utsname.h>**.

The *uname*( ) function shall return a string naming the current system in the character array *sysname*. Similarly, *nodename* shall contain the name of this node within an implementation-defined communications network. The arrays *release* and *version* shall further identify the operating system. The array *machine* shall contain a name that identifies the hardware that the system is running on.

The format of each member is implementation-defined.

RETURN VALUE

Upon successful completion, a non-negative value shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The inclusion of the *nodename* member in this structure does not imply that it is sufficient information for interfacing to communications networks.

RATIONALE

The values of the structure members are not constrained to have any relation to the version of this volume of IEEE Std 1003.1-200x implemented in the operating system. An application should instead depend on _POSIX_VERSION and related constants defined in **<unistd.h>**.

This volume of IEEE Std 1003.1-200x does not define the sizes of the members of the structure and permits them to be of different sizes, although most implementations define them all to be the same size: eight bytes plus one byte for the string terminator. That size for *nodename* is not enough for use with many networks.

The *uname*( ) function originated in System III, System V, and related implementations, and it does not exist in Version 7 or 4.3 BSD. The values it returns are set at system compile time in those historical implementations.

4.3 BSD has *gethostname*( ) and *gethostid*( ), which return a symbolic name and a numeric value, respectively. There are related *sethostname*( ) and *sethostid*( ) functions that are used to set the values the other two functions return. The former functions are included in this specification, the latter are not.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/utsname.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

ungetc — push byte back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ungetc*( ) function shall push the byte specified by *c* (converted to an **unsigned char**) back onto the input stream pointed to by *stream*. The pushed-back bytes shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )) shall discard any pushed-back bytes for the stream. The external storage corresponding to the stream shall be unchanged.

One byte of push-back shall be provided. If *ungetc*( ) is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of *c* equals that of the macro EOF, the operation shall fail and the input stream shall be left unchanged.

A successful call to *ungetc*( ) shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back bytes shall be the same as it was before the bytes were pushed back. The file-position indicator is decremented by each successful call to *ungetc*( ); if its value was 0 before a call, its value is unspecified after the call.

RETURN VALUE

Upon successful completion, *ungetc*( ) shall return the byte pushed back after conversion. Otherwise, it shall return EOF.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fseek( ), *getc*( ), *fsetpos*( ), *read*( ), *rewind*( ), *setbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

NAME

ungetwc — push wide-character code back into input stream

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

wint_t ungetwc(wint_t wc, FILE *stream);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *ungetwc*( ) function shall push the character corresponding to the wide-character code specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back characters shall be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file-positioning function (*fseek*( ), *fsetpos*( ), or *rewind*( )) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

At least one character of push-back shall be provided. If *ungetwc*( ) is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail. |

If the value of *wc* equals that of the macro WEOF, the operation shall fail and the input stream shall be left unchanged. |

A successful call to *ungetwc*( ) shall clear the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. The file-position indicator is decremented (by one or more) by each successful call to *ungetwc*( ); if its value was 0 before a call, its value is unspecified after the call. |

RETURN VALUE

Upon successful completion, *ungetwc*( ) shall return the wide-character code corresponding to the pushed-back character. Otherwise, it shall return WEOF.

ERRORS

The *ungetwc*( ) function may fail if:

CX [EILSEQ] An invalid character sequence is detected, or a wide-character code does not correspond to a valid character. |

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fseek( ), *fsetpos*( ), *read*( ), *rewind*( ), *setbuf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The Optional Header (OH) marking is removed from **<stdio.h>**.

Issue 6

The [EILSEQ] optional error condition is marked CX.

NAME

unlink — remove a directory entry

SYNOPSIS

```
#include <unistd.h>

int unlink(const char *path);
```

DESCRIPTION

The *unlink*( ) function shall remove a link to a file. If *path* names a symbolic link, *unlink*( ) shall remove the symbolic link named by *path* and shall not affect any file or directory named by the contents of the symbolic link. Otherwise, *unlink*( ) shall remove the link named by the pathname pointed to by *path* and shall decrement the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *unlink*( ) returns, but the removal of the file contents shall be postponed until all references to the file are closed.

The *path* argument shall not name a directory unless the process has appropriate privileges and the implementation supports using *unlink*( ) on directories.

Upon successful completion, *unlink*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the parent directory. Also, if the file's link count is not 0, the *st_ctime* field of the file shall be marked for update.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error. If −1 is returned, the named file shall not be changed.

ERRORS

The *unlink*( ) function shall fail and shall not unlink the file if:

[EACCES]	Search permission is denied for a component of the path prefix, or write permission is denied on the directory containing the directory entry to be removed.
[EBUSY]	The file named by the *path* argument cannot be unlinked because it is being used by the system or another process and the implementation considers this an error.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The file named by *path* is a directory, and either the calling process does not have appropriate privileges, or the implementation prohibits using *unlink*( ) on directories.
XSI [EPERM] or [EACCES]	The S_ISVTX flag is set on the directory containing the file referred to by the *path* argument and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

[EROFS]	The directory entry to be unlinked is part of a read-only file system.

The *unlink*( ) function may fail and not unlink the file if:

XSI [EBUSY]	The file named by *path* is a named STREAM.
[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.
[ETXTBSY]	The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

EXAMPLES

Removing a Link to a File

The following example shows how to remove a link to a file named **/home/cnd/mod1** by removing the entry named **/modules/pass1**.

```
#include <unistd.h>

char *path = "/modules/pass1";
int   status;
...
status = unlink(path);
```

Checking for an Error

The following example fragment creates a temporary password lock file named **LOCKFILE**, which is defined as **/etc/ptmp**, and gets a file descriptor for it. If the file cannot be opened for writing, *unlink*( ) is used to remove the link between the file descriptor and **LOCKFILE**.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

#define LOCKFILE "/etc/ptmp"

int pfd;  /* Integer for file descriptor returned by open call. */
FILE *fpfd;  /* File pointer for use in putpwent(). */
...
/* Open password Lock file. If it exists, this is an error. */
if ((pfd = open(LOCKFILE, O_WRONLY| O_CREAT | O_EXCL, S_IRUSR
    | S_IWUSR | S_IRGRP | S_IROTH)) == -1)  {
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n");
    exit(1);
}

/* Lock file created, proceed with fdopen of lock file so that
   putpwent() can be used.
  */
if ((fpfd = fdopen(pfd, "w")) == NULL) {
```

```
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
```

Replacing Files

The following example fragment uses *unlink*( ) to discard links to files, so that they can be replaced with new versions of the files. The first call remove the link to **LOCKFILE** if an error occurs. Successive calls remove the links to **SAVEFILE** and **PASSWDFILE** so that new links can be created, then removes the link to **LOCKFILE** when it is no longer needed.

```
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
#define SAVEFILE "/etc/opasswd"
...
/* If no change was made, assume error and leave passwd unchanged. */
if (!valid_change) {
    fprintf(stderr, "Could not change password for user %s\n", user);
    unlink(LOCKFILE);
    exit(1);
}

/* Change permissions on new password file. */
chmod(LOCKFILE, S_IRUSR | S_IRGRP | S_IROTH);

/* Remove saved password file. */
unlink(SAVEFILE);

/* Save current password file. */
link(PASSWDFILE, SAVEFILE);

/* Remove current password file. */
unlink(PASSWDFILE);

/* Save new password file as current password file. */
link(LOCKFILE,PASSWDFILE);

/* Remove lock file. */
unlink(LOCKFILE);

exit(0);
```

APPLICATION USAGE

Applications should use *rmdir*( ) to remove a directory.

RATIONALE

Unlinking a directory is restricted to the superuser in many historical implementations for reasons given in *link*( ) (see also *rename*( )).

The meaning of [EBUSY] in historical implementations is ‘‘mount point busy’’. Since this volume of IEEE Std 1003.1-200x does not cover the system administration concepts of mounting and unmounting, the description of the error was changed to ‘‘resource busy’’. (This meaning is used by some device drivers when a second process tries to open an exclusive use device.) The wording is also intended to allow implementations to refuse to remove a directory if it is the root or current working directory of any process.

FUTURE DIRECTIONS

None.

SEE ALSO

close( ), *link*( ), *remove*( ), *rmdir*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The [EBUSY] error is added to the ‘‘may fail’’ part of the ERRORS section.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the effect is specified if *path* specifies a symbolic link.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

unlockpt — unlock a pseudo-terminal master/slave pair

SYNOPSIS

XSI
```
#include <stdlib.h>

int unlockpt(int fildes);
```

DESCRIPTION

The *unlockpt*( ) function shall unlock the slave pseudo-terminal device associated with the master to which *fildes* refers.

Conforming applications shall ensure that they call *unlockpt*( ) before opening the slave side of a pseudo-terminal device. |

RETURN VALUE

Upon successful completion, *unlockpt*( ) shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *unlockpt*( ) function may fail if:

[EBADF]	The *fildes* argument is not a file descriptor open for writing.
[EINVAL]	The *fildes* argument is not associated with a master pseudo-terminal device.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

grantpt( ), *open*( ), *ptsname*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

unsetenv — remove environment variable

SYNOPSIS

CX

```
#include <stdlib.h>

int unsetenv(const char *name);
```

DESCRIPTION

The *unsetenv*( ) function shall remove an environment variable from the environment of the calling process. The *name* argument points to a string, which is the name of the variable to be removed. The named argument shall not contain an `'='` character. If the named variable does not exist in the current environment, the environment shall be unchanged and the function is considered to have completed successfully.

If the application modifies *environ* or the pointers to which it points, the behavior of *unsetenv*( ) is undefined. The *unsetenv*( ) function shall update the list of pointers to which *environ* points.

The *unsetenv*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

Upon successful completion, zero shall be returned. Otherwise, −1 shall be returned, *errno* set to indicate the error, and the environment shall be unchanged.

ERRORS

The *unsetenv*( ) function shall fail if:

[EINVAL]	The *name* argument is a null pointer, points to an empty string, or points to a string containing an `'='` character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

Refer to the RATIONALE section in *setenv*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

getenv( ), *setenv*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**, **<sys/types.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the IEEE P1003.1a draft standard.

NAME

usleep — suspend execution for an interval

SYNOPSIS

OB XSI

```
#include <unistd.h>

int usleep(useconds_t useconds);
```

DESCRIPTION

The *usleep*( ) function shall cause the calling thread to be suspended from execution until either the number of realtime microseconds specified by the argument *useconds* has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

The *useconds* argument shall be less than one million. If the value of *useconds* is 0, then the call has no effect. |

If a SIGALRM signal is generated for the calling process during execution of *usleep*( ) and if the SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether *usleep*( ) returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after *usleep*( ) returns or it is discarded.

If a SIGALRM signal is generated for the calling process during execution of *usleep*( ), except as a result of a prior call to *alarm*( ), and if the SIGALRM signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing *usleep*( ) to return.

If a signal-catching function interrupts *usleep*( ) and examines or changes either the time a SIGALRM is scheduled to be generated, the action associated with the SIGALRM signal, or whether the SIGALRM signal is blocked from delivery, the results are unspecified.

If a signal-catching function interrupts *usleep*( ) and calls *siglongjmp*( ) or *longjmp*( ) to restore an environment saved prior to the *usleep*( ) call, the action associated with the SIGALRM signal and the time at which a SIGALRM signal is scheduled to be generated are unspecified. It is also unspecified whether the SIGALRM signal is blocked, unless the process' signal mask is restored as part of the environment.

Implementations may place limitations on the granularity of timer values. For each interval timer, if the requested timer value requires a finer granularity than the implementation supports, the actual timer value shall be rounded up to the next supported value.

Interactions between *usleep*( ) and any of the following are unspecified:

nanosleep( )
setitimer( )
timer_create( )
timer_delete( )
timer_getoverrun( )
timer_gettime( )
timer_settime( )
ualarm( )
sleep( )

RETURN VALUE

Upon successful completion, *usleep*( ) shall return 0; otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

The *usleep*( ) function may fail if:

[EINVAL]	The time interval specified one million or more microseconds.

EXAMPLES

None.

APPLICATION USAGE

Applications are recommended to use *nanosleep*( ) if the Timers option is supported, or *setitimer*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), *timer_gettime*( ), or *timer_settime*( ) instead of this function.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *getitimer*( ), *nanosleep*( ), *sigaction*( ), *sleep*( ), *timer_create*( ), *timer_delete*( ), *timer_getoverrun*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is changed to indicate that timers are now thread-based rather than process-based.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

This function is marked obsolescent.

NAME

utime — set file access and modification times

SYNOPSIS

```
#include <utime.h>

int utime(const char *path, const struct utimbuf *times);
```

DESCRIPTION

The *utime*( ) function shall set the access and modification times of the file named by the *path* argument.

If *times* is a null pointer, the access and modification times of the file shall be set to the current time. The effective user ID of the process shall match the owner of the file, or the process has write permission to the file or has appropriate privileges, to use *utime*( ) in this manner.

If *times* is not a null pointer, *times* shall be interpreted as a pointer to a **utimbuf** structure and the access and modification times shall be set to the values contained in the designated structure. Only a process with effective user ID equal to the user ID of the file or a process with appropriate privileges may use *utime*( ) this way.

The **utimbuf** structure is defined in the **<utime.h>** header. The times in the structure **utimbuf** are measured in seconds since the Epoch.

Upon successful completion, *utime*( ) shall mark the time of the last file status change, *st_ctime*, to be updated; see **<sys/stat.h>**.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* shall be set to indicate the error, and the file times shall not be affected.

ERRORS

The *utime*( ) function shall fail if:

[EACCES]	Search permission is denied by a component of the path prefix; or the *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file, the process does not have write permission for the file, and the process does not have appropriate privileges.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The *times* argument is not a null pointer and the calling process' effective user ID does not match the owner of the file and the calling process does not have the appropriate privileges.
[EROFS]	The file system containing the file is read-only.

The *utime*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.

[ENAMETOOLONG]
As a result of encountering a symbolic link in resolution of the *path* argument, the length of the substituted pathname string exceeded {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

The *actime* structure member must be present so that an application may set it, even though an implementation may ignore it and not change the access time on the file. If an application intends to leave one of the times of a file unchanged while changing the other, it should use *stat*( ) to retrieve the file's *st_atime* and *st_mtime* parameters, set *actime* and *modtime* in the buffer, and change one of them before making the *utime*( ) call.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<utime.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [ELOOP] optional error condition is added.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

utimes — set file access and modification times (**LEGACY**)

SYNOPSIS

XSI

```
#include <sys/time.h>

int utimes(const char *path, const struct timeval times[2]);
```

DESCRIPTION

The *utimes*( ) function shall set the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. The *utimes*( ) function allows time specifications accurate to the microsecond.

For *utimes*( ), the *times* argument is an array of **timeval** structures. The first array member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the **timeval** structure are measured in seconds and microseconds since the Epoch, although rounding toward the nearest second may occur.

If the *times* argument is a null pointer, the access and modification times of the file shall be set to the current time. The effective user ID of the process shall match the owner of the file, or has write access to the file or appropriate privileges to use this call in this manner. Upon completion, *utimes*( ) shall mark the time of the last file status change, *st_ctime*, for update.

RETURN VALUE

Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and *errno* shall be set to indicate the error, and the file times shall not be affected.

ERRORS

The *utimes*( ) function shall fail if:

[EACCES]	Search permission is denied by a component of the path prefix; or the *times* argument is a null pointer and the effective user ID of the process does not match the owner of the file and write access is denied.
[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.
[ENAMETOOLONG]	The length of the *path* argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.
[ENOENT]	A component of *path* does not name an existing file or *path* is an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The *times* argument is not a null pointer and the calling process' effective user ID has write access to the file but does not match the owner of the file and the calling process does not have the appropriate privileges.
[EROFS]	The file system containing the file is read-only.

The *utimes*( ) function may fail if:

[ELOOP]	More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the *path* argument.
[ENAMETOOLONG]	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

EXAMPLES

None.

APPLICATION USAGE

For applications portability, the *utime*( ) function should be used to set file access and modification times instead of *utimes*( ).

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<sys/time.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

This function is marked LEGACY.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements. |

The wording of the mandatory [ELOOP] error condition is updated, and a second optional [ELOOP] error condition is added.

NAME

va_arg, va_copy, va_end, va_start — handle variable argument list

SYNOPSIS

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

DESCRIPTION

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**.

NAME

vfork — create new process; share virtual memory

SYNOPSIS

OB XSI

```
#include <unistd.h>

pid_t vfork(void);
```

DESCRIPTION

The *vfork*( ) function shall be equivalent to *fork*( ), except that the behavior is undefined if the process created by *vfork*( ) either modifies any data other than a variable of type **pid_t** used to store the return value from *vfork*( ), or returns from the function in which *vfork*( ) was called, or calls any other function before successfully calling *_exit*( ) or one of the *exec* family of functions.

RETURN VALUE

Upon successful completion, *vfork*( ) shall return 0 to the child process and return the process ID of the child process to the parent process. Otherwise, –1 shall be returned to the parent, no child process shall be created, and *errno* shall be set to indicate the error.

ERRORS

The *vfork*( ) function shall fail if:

[EAGAIN]	The system-wide limit on the total number of processes under execution would be exceeded, or the system-imposed limit on the total number of processes under execution by a single user would be exceeded.
[ENOMEM]	There is insufficient swap space for the new process.

EXAMPLES

None.

APPLICATION USAGE

Conforming applications are recommended not to depend on *vfork*( ), but to use *fork*( ) instead. The *vfork*( ) function may be withdrawn in a future version.

On some implementations, *vfork*( ) is equivalent to *fork*( ).

The *vfork*( ) function differs from *fork*( ) only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if *vfork*( ) is misused.

The use of *vfork*( ) for any purpose except as a prelude to an immediate call to a function from the *exec* family, or to *_exit*( ), is not advised.

The *vfork*( ) function can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to call *exec*, the data space copied from the parent to the child by *fork*( ) is not used. This is particularly inefficient in a paged environment, making *vfork*( ) particularly useful. Depending upon the size of the parent's data space, *vfork*( ) can give a significant performance improvement over *fork*( ).

The *vfork*( ) function can normally be used just like *fork*( ). It does not work, however, to return while running in the child's context from the caller of *vfork*( ) since the eventual return from *vfork*( ) would then return to a no longer existent stack frame. Care should be taken, also, to call *_exit*( ) rather than *exit*( ) if *exec* cannot be used, since *exit*( ) flushes and closes standard I/O channels, thereby damaging the parent process' standard I/O data structures. (Even with *fork*( ), it is wrong to call *exit*( ), since buffered data would then be flushed twice.)

If signal handlers are invoked in the child process after *vfork*( ), they must follow the same rules as other code in the child process.

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

exec, *exit*( ), *fork*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<unistd.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

Issue 6

Marked obsolescent.

NAME

vfprintf, vprintf, vsnprintf, vsprintf — format output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(FILE *restrict stream, const char *restrict format,
    va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
    va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_list ap);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *vprintf*( ), *vfprintf*( ), *vsnprintf*( ), and *vsprintf*( ) functions shall be equivalent to *printf*( ), *fprintf*( ), *snprintf*( ), and *sprintf*( ) respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **<stdarg.h>**.

These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

RETURN VALUE

Refer to *fprintf*( ).

ERRORS

Refer to *fprintf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fprintf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The *vsnprintf*( ) function is added.

Issue 6

The *vfprintf*( ), *vprintf*( ), *vsnprintf*( ), and *vsprintf*( ) functions are updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

vfscanf, vscanf, vsscanf — format input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vfscanf(FILE *restrict stream, const char *restrict format,
    va_list arg);
int vscanf(const char *restrict format, va_list arg);
int vsscanf(const char *restrict s, const char *restrict format,
    va_list arg);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *vscanf*( ), *vfscanf*( ), and *vsscanf*( ) functions shall be equivalent to the *scanf*( ), *fscanf*( ), and *sscanf*( ) functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the **<stdarg.h>** header. These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

RETURN VALUE

Refer to *fscanf*( ).

ERRORS

Refer to *fscanf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fscanf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

vfwprintf, vswprintf, vwprintf — wide-character formatted output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
    va_list arg);
int vswprintf(wchar_t *restrict ws, size_t n,
    const wchar_t *restrict format, va_list arg);
int vwprintf(const wchar_t *restrict format, va_list arg);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *vfwprintf*( ), *vswprintf*( ), and *vwprintf*( ) functions shall be equivalent to *fwprintf*( ), *swprintf*( ), and *wprintf*( ) respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **<stdarg.h>**.

These functions shall not invoke the *va_end* macro. However, as these functions do invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

RETURN VALUE

Refer to *fwprintf*( ).

ERRORS

Refer to *fwprintf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fwprintf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *vfwprintf*( ), *vswprintf*( ), and *vwprintf*( ) prototypes are updated for alignment with the ISO/IEC 9899:1999 standard. ( )

NAME

vfwscanf, vswscanf, vwscanf — wide-character formattted input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
    va_list arg);
int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
    va_list arg);
int vwscanf(const wchar_t *restrict format, va_list arg);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *vfwscanf*( ), *vswscanf*( ), and *vwscanf*( ) functions shall be equivalent to the *fwscanf*( ), *swscanf*( ), and *wscanf*( ) functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the **<stdarg.h>** header. These functions shall not invoke the *va_end* macro. As these functions invoke the *va_arg* macro, the value of *ap* after the return is unspecified.

RETURN VALUE

Refer to *fwscanf*( ).

ERRORS

Refer to *fwscanf*( ).

EXAMPLES

None.

APPLICATION USAGE

Applications using these functions should call *va_end*(*ap*) afterwards to clean up.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fwscanf( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdarg.h>**, **<stdio.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

vprintf — format output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *restrict format, va_list ap);
```

DESCRIPTION

Refer to *vfprintf*( ).

NAME

vscanf — format input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vscanf(const char *restrict format, va_list arg);
```

DESCRIPTION

Refer to *vfscanf*( ).

NAME

vsnprintf, vsprintf — format output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vsnprintf(char *restrict s, size_t n,
    const char *restrict format, va_list ap);
int vsprintf(char *restrict s, const char *restrict format,
    va_list ap);
```

DESCRIPTION

Refer to *vfprintf*( ).

NAME

vsscanf — format input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vsscanf(const char *restrict s, const char *restrict format,
    va_list arg);
```

DESCRIPTION

Refer to *vfscanf*( ).

NAME

vswprintf — wide-character formatted output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vswprintf(wchar_t *restrict ws, size_t n,
    const wchar_t *restrict format, va_list arg);
```

DESCRIPTION

Refer to *vfwprintf*( ).

NAME

vswscanf — wide-character formattted input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vswscanf(const wchar_t *restrict ws, const wchar_t *restrict format,
    va_list arg);
```

DESCRIPTION

Refer to *vfwscanf*( ).

NAME

vwprintf — wide-character formatted output of a stdarg argument list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vwprintf(const wchar_t *restrict format, va_list arg);
```

DESCRIPTION

Refer to *vfwprintf*( ).

NAME

vwscanf — wide-character formattted input of a stdarg list

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vwscanf(const wchar_t *restrict format, va_list arg);
```

DESCRIPTION

Refer to *vfwscanf*( ).

NAME

wait, waitpid — wait for a child process to stop or terminate

SYNOPSIS

```
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

The *wait*( ) and *waitpid*( ) functions shall obtain status information pertaining to one of the caller's child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The *wait*( ) function shall suspend execution of the calling thread until status information for one of the terminated child processes of the calling process is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in *wait*( ) or *waitpid*( ) awaiting termination of the same process, exactly one thread shall return the process status at the time of the target process termination. If status information is available prior to the call to *wait*( ), return shall be immediate.

The *waitpid*( ) function shall be equivalent to *wait*( ) if the *pid* argument is (**pid_t**)−1 and the *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid*( ) function shall only return the status of a child process from this set:

- If *pid* is equal to (**pid_t**)−1, *status* is requested for any child process. In this respect, *waitpid*( ) is then equivalent to *wait*( ).
- If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is requested.
- If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of the calling process.
- If *pid* is less than (**pid_t**)−1, *status* is requested for any child process whose process group ID is equal to the absolute value of *pid*.

The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the **<sys/wait.h>** header:

XSI	WCONTINUED	The *waitpid*( ) function shall report the status of any continued child process specified by *pid* whose status has not been reported since it continued from a job control stop.
	WNOHANG	The *waitpid*( ) function shall not suspend execution of the calling thread if *status* is not immediately available for one of the child processes specified by *pid*.
	WUNTRACED	The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

XSI If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited-for children that were transformed into zombie processes, the calling thread shall block until all of the children of the process containing the calling thread terminate, and *wait*( ) and *waitpid*( ) shall fail and set *errno* to [ECHILD].

If *wait*( ) or *waitpid*( ) return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not a null pointer, information shall be stored in the location pointed to by *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from *main*( ).
2. The process called *_exit*( ) or *exit*( ) with a *status* argument of 0.
3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in **<sys/wait.h>** and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)
Evaluates to a non-zero value if *status* was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)
If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to *_exit*( ) or *exit*( ), or the value the child process returned from *main*( ).

WIFSIGNALED(*stat_val*)
Evaluates to non-zero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught (see **<signal.h>**).

WTERMSIG(*stat_val*)
If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)
Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

XSI WIFCONTINUED(*stat_val*)
Evaluates to a non-zero value if status was returned for a child process that has continued from a job control stop.

SPN It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes created by *posix_spawn*( ) or *posix_spawnp*( ) can indicate a WIFSTOPPED(*stat_val*) before subsequent calls to *wait*( ) or *waitpid*( ) indicate WIFEXITED(*stat_val*) as the result of an error detected before the new process image starts executing.

It is unspecified whether the *status* value returned by calls to *wait*( ) or *waitpid*( ) for processes created by *posix_spawn*( ) or *posix_spawnp*( ) can indicate a WIFSIGNALED(*stat_val*) if a signal is sent to the parent's process group after *posix_spawn*( ) or *posix_spawnp*( ) is called.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that specified the XSI WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) shall evaluate to a non-zero value.

XSI If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that specified the WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

XSI If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that did not specify the WUNTRACED or WCONTINUED flags, or by a call to the *wait*( ) function, exactly one of the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero value.

XSI If the information pointed to by *stat_loc* was stored by a call to *waitpid*( ) that did not specify the WUNTRACED flag and specified the WCONTINUED flag, or by a call to the *wait*( ) function, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD signal, then if *wait*( ) or *waitpid*( ) returns because the status of a child process is available, any pending SIGCHLD signal associated with the process ID of the child process shall be discarded. Any other pending SIGCHLD signals shall remain pending.

Otherwise, if SIGCHLD is blocked, if *wait*( ) or *waitpid*( ) return because the status of a child process is available, any pending SIGCHLD signal shall be cleared unless the status of another child process is available.

For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD signal is delivered.

There may be additional implementation-defined circumstances under which *wait*( ) or *waitpid*( ) report *status*. This shall not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported *status* is implementation-defined.

XSI If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

RETURN VALUE

If *wait*( ) or *waitpid*( ) returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which *status* is reported. If *wait*( ) or *waitpid*( ) returns due to the delivery of a signal to the calling process, −1 shall be returned and *errno* set to [EINTR]. If *waitpid*( ) was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which *status* is not available, and *status* is not available for any process specified by *pid*, 0 is returned. Otherwise, (**pid_t**)−1 shall be returned, and *errno* set to indicate the error.

ERRORS

The *wait*( ) function shall fail if:

[ECHILD]	The calling process has no existing unwaited-for child processes.
[EINTR]	The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

The *waitpid*( ) function shall fail if:

[ECHILD]	The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid* does not exist or does not have any member process that is a child of the calling process.

[EINTR]	The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.
[EINVAL]	The *options* argument is not valid.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

A call to the *wait*( ) or *waitpid*( ) function only returns *status* on an immediate child process of the calling process; that is, a child that was produced by a single *fork*( ) call (perhaps followed by an *exec* or other function calls) from the parent. If a child produces grandchildren by further use of *fork*( ), none of those grandchildren nor any of their descendants affect the behavior of a *wait*( ) from the original parent process. Nothing in this volume of IEEE Std 1003.1-200x prevents an implementation from providing extensions that permit a process to get *status* from a grandchild or any other process, but a process that does not use such extensions must be guaranteed to see *status* from only its direct children.

The *waitpid*( ) function is provided for three reasons:

1. To support job control
2. To permit a non-blocking version of the *wait*( ) function
3. To permit a library routine, such as *system*( ) or *pclose*( ), to wait for its children without interfering with other terminated children for which the process has not waited

The first two of these facilities are based on the *wait3*( ) function provided by 4.3 BSD. The function uses the *options* argument, which is equivalent to an argument to *wait3*( ). The WUNTRACED flag is used only in conjunction with job control on systems supporting job control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped processes in that implementation: processes being traced via the *ptrace*( ) debugging facility and (untraced) processes stopped by job control signals. Since *ptrace*( ) is not part of this volume of IEEE Std 1003.1-200x, only the second type is relevant. The name WUNTRACED was retained because its usage is the same, even though the name is not intuitively meaningful in this context.

The third reason for the *waitpid*( ) function is to permit independent sections of a process to spawn and wait for children without interfering with each other. For example, the following problem occurs in developing a portable shell, or command interpreter:

```
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all historical implementations, the final *pclose*( ) fails to reap the *wait*( ) *status* of the *popen*( ).

The status values are retrieved by macros, rather than given as specific bit encodings as they are in most historical implementations (and thus expected by existing programs). This was necessary to eliminate a limitation on the number of signals an implementation can support that was inherent in the traditional encodings. This volume of IEEE Std 1003.1-200x does require that a *status* value of zero corresponds to a process calling *_exit*(0), as this is the most common encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

These macros syntactically operate on an arbitrary integer value. The behavior is undefined unless that value is one stored by a successful call to *wait*( ) or *waitpid*( ) in the location pointed to by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other specifications in this volume of IEEE Std 1003.1-200x or traditional usage. It also could have implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects *wait*( ) and *waitpid*( ) and is common in historical implementations is the *ptrace*( ) function. It is called by a child process and causes that child to stop and return a *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace*( ) children is traditionally returned regardless of the WUNTRACED flag (or by the *wait*( ) function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support *core* file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the *status* returned by *wait*( ) to indicate that such actions have occurred.

Allowing the *wait*( ) family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the *sigwait*( ) and *sigwaitinfo*( ) category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in *wait*( ), with the same meaning of ‘‘accepted’’ as when that word is applied to the *sigwait*( ) family of functions.

Allowing the *wait*( ) family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via *sigaction*( ) without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.

2. A single *wait*( ) issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.

3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-NULL pointer to a **siginfo_t** structure that describes a child process for which a wait via *waitpid*( ) or *waitid*( ) will not block or fail.

4. The *system*( ) function will not cause a processs SIGCHLD handler to be called as a result of the *fork*( )/*exec* executed within *system*( ) because *system*( ) will accept the SIGCHLD signal when it performs a *waitpid*( ) for its child process. This is a desirable behavior of *system*( ) so that it can be used in a library without causing side effects to the application linked with the library.

An implementation that does not permit the *wait*( ) family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1

Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter example is the case of a process that blocks SIGCHLD and performs an indefinite loop of *fork*( )/*wait*( ) operations. If the implementation supports queued signals, then eventually the system will run out of memory for the queue. The guarantee cannot be made because there must be some limit to the depth of queuing.

Guarantees #2 and #3

These cannot be guaranteed unless the *wait*( ) family of functions accepts the SIGCHLD signal. Otherwise, a *fork*( )/*wait*( ) executed while SIGCHLD is blocked (as in the *system*( ) function) will result in an invocation of the handler when SIGCHLD is unblocked, after the process has disappeared.

Guarantee #4

Although possible to make this guarantee, *system*( ) would have to set the SIGCHLD handler to SIG_DFL so that the SIGCHLD signal generated by its *fork*( ) would be discarded (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This would have the undesirable side effect of discarding all SIGCHLD signals pending to the process.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *fork*( ), *waitid*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/types.h>**, **<sys/wait.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

In the SYNOPSIS, the optional include of the **<sys/types.h>** header is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The requirement to include **<sys/types.h>** has been removed. Although **<sys/types.h>** was required for conforming implementations of previous POSIX specifications, it was not required for UNIX applications.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The processing of the SIGCHLD signal and the [ECHILD] error is clarified.

The semantics of WIFSTOPPED(*stat_val*), WIFEXITED(*stat_val*), and WIFSIGNALED(*stat_val*) are defined with respect to *posix_spawn*( ) or *posix_spawnp*( ) for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

waitid — wait for a child process to change state

SYNOPSIS

XSI

```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

DESCRIPTION

The *waitid*( ) function shall suspend the calling thread until one child of the process containing the calling thread changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call to *waitid*( ), *waitid*( ) shall return immediately. If more than one thread is suspended in *wait*( ) or *waitpid*( ) waiting termination of the same process, exactly one thread shall return the process status at the time of the target process termination.

The *idtype* and *id* arguments are used to specify which children *waitid*( ) waits for.

If *idtype* is P_PID, *waitid*( ) shall wait for the child with a process ID equal to (**pid_t**)*id*.

If *idtype* is P_PGID, *waitid*( ) shall wait for any child with a process group ID equal to (**pid_t**)*id*.

If *idtype* is P_ALL, *waitid*( ) shall wait for any children and *id* is ignored.

The *options* argument is used to specify which state changes *waitid*( ) shall wait for. It is formed by OR'ing together one or more of the following flags:

WEXITED	Wait for processes that have exited.
WSTOPPED	Status shall be returned for any child that has stopped upon receipt of a signal.
WCONTINUED	Status shall be returned for any child that was stopped and has been continued.
WNOHANG	Return immediately if there are no children to wait for.
WNOWAIT	Keep the process whose status is returned in *infop* in a waitable state. This shall not affect the state of the process; the process may be waited for again after this call completes.

The application shall ensure that the *infop* argument points to a **siginfo_t** structure. If *waitid*( ) returns because a child process was found that satisfied the conditions indicated by the arguments *idtype* and *options*, then the structure pointed to by *infop* shall be filled in by the system with the status of the process. The *si_signo* member shall always be equal to SIGCHLD.

RETURN VALUE

If WNOHANG was specified and there are no children to wait for, 0 shall be returned. If *waitid*( ) returns due to the change of state of one of its children, 0 shall be returned. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

The *waitid*( ) function shall fail if:

[ECHILD]	The calling process has no existing unwaited-for child processes.
[EINTR]	The *waitid*( ) function was interrupted by a signal.
[EINVAL]	An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *exit*( ), *wait*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<sys/wait.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 5

Moved from X/OPEN UNIX extension to BASE.

The DESCRIPTION is updated for alignment with the POSIX Threads Extension.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

waitpid — wait for a child process to stop or terminate

SYNOPSIS

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION

Refer to *wait*( ).

NAME

wcrtomb — convert a wide-character code to a character (restartable)

SYNOPSIS

```
#include <stdio.h>

size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

If *s* is a null pointer, the *wcrtomb*( ) function shall be equivalent to the call:

```
wcrtomb(buf, L'\0', ps)
```

where *buf* is an internal buffer.

If *s* is not a null pointer, the *wcrtomb*( ) function shall determine the number of bytes needed to represent the character that corresponds to the wide character given by *wc* (including any shift sequences), and store the resulting bytes in the array whose first element is pointed to by *s*. At most {MB_CUR_MAX} bytes are stored. If *wc* is a null wide character, a null byte shall be stored, preceded by any shift sequence needed to restore the initial shift state. The resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *wcrtomb*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *wcrtomb*( ).

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcrtomb*( ) function is called with a non-NULL *ps* argument.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *wcrtomb*( ) function shall return the number of bytes stored in the array object (including any shift sequences). When *wc* is not a valid wide character, an encoding error shall occur. In this case, the function shall store the value of the macros [EILSEQ] in *errno* and shall return (**size_t**)−1; the conversion state shall be undefined.

ERRORS

The *wcrtomb*( ) function may fail if:

CX	[EINVAL]	*ps* points to an object that contains an invalid conversion state.
	[EILSEQ]	Invalid wide-character code is detected.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

In the DESCRIPTION, a note on using this function in a threaded application is added.

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *wcrtomb*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcscat — concatenate two wide-character strings

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscat(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscat*( ) function shall append a copy of the wide-character string pointed to by *ws2* (including the terminating null wide-character code) to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. If copying takes place between objects that overlap, the behavior is undefined. |

RETURN VALUE

The *wcscat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The Open Group Corrigendum U040/2 is applied. In the RETURN VALUE section, *s1* is changed to *ws1*.

The *wcscat*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcschr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *ws, wchar_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcschr*( ) function shall locate the first occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

RETURN VALUE

Upon completion, *wcschr*( ) shall return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsrchr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wcscmp — compare two wide-character strings

SYNOPSIS

```
#include <wchar.h>

int wcscmp(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscmp*( ) function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. |

RETURN VALUE

Upon completion, *wcscmp*( ) shall return an integer greater than, equal to, or less than 0, if the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcscoll — wide-character string comparison using collating information

SYNOPSIS

```
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscoll*( ) function shall compare the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, both interpreted as appropriate to the *LC_COLLATE* category of the current locale.

CX The *wcscoll*( ) function shall not change the setting of *errno* if successful.

An application wishing to check for error situations should set *errno* to 0 before calling *wcscoll*( ). If *errno* is non-zero on return, an error has occurred.

RETURN VALUE

Upon successful completion, *wcscoll*( ) shall return an integer greater than, equal to, or less than 0, according to whether the wide-character string pointed to by *ws1* is greater than, equal to, or less than the wide-character string pointed to by *ws2*, when both are interpreted as appropriate to the current locale. CX On error, *wcscoll*( ) shall set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *wcscoll*( ) function may fail if:

CX [EINVAL] The *ws1* or *ws2* arguments contain wide-character codes outside the domain of the collating sequence.

EXAMPLES

None.

APPLICATION USAGE

The *wcsxfrm*( ) and *wcscmp*( ) functions should be used for sorting large lists.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), *wcsxfrm*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

NAME

wcscpy — copy a wide-character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscpy*( ) function shall copy the wide-character string pointed to by *ws2* (including the terminating null wide-character code) into the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcscpy*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsncpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The *wcscpy*( ) prototype is updated for alignment with the ISO/IEC 9899: 1999 standard.

NAME

wcscspn — get length of a complementary wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcscspn(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcscspn*( ) function shall compute the length (in wide characters) of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes *not* from the wide-character string pointed to by *ws2*. |

RETURN VALUE

The *wcscspn*( ) function shall return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcsspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The RETURN VALUE section is updated to indicate that *wcscspn*( ) returns the length of *ws1*, rather than *ws1* itself.

NAME

wcsftime — convert date and time to a wide-character string

SYNOPSIS

```
#include <wchar.h>

size_t wcsftime(wchar_t *restrict wcs, size_t maxsize,
    const wchar_t *restrict format, const struct tm *restrict timeptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsftime*( ) function shall be equivalent to the *strftime*( ) function, except that:

- The argument *wcs* points to the initial element of an array of wide characters into which the generated output is to be placed.
- The argument *maxsize* indicates the maximum number of wide characters to be placed in the output array.
- The argument *format* is a wide-character string and the conversion specifications are replaced by corresponding sequences of wide characters.
- The return value indicates the number of wide characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

If the total number of resulting wide-character codes including the terminating null wide-character code is no more than *maxsize*, *wcsftime*( ) shall return the number of wide-character codes placed into the array pointed to by *wcs*, not including the terminating null wide-character code. Otherwise, zero is returned and the contents of the array are unspecified.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

strftime( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, the type of the *format* argument is changed from **const char *** to **const wchar_t ***.

Issue 6

The *wcsftime*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcslen — get wide-character string length

SYNOPSIS

```
#include <wchar.h>

size_t wcslen(const wchar_t *ws);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcslen*( ) function shall compute the number of wide-character codes in the wide-character string to which *ws* points, not including the terminating null wide-character code.

RETURN VALUE

The *wcslen*( ) function shall return the length of *ws*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcsncat — concatenate a wide-character string with part of another

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsncat*( ) function shall append not more than *n* wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by *ws2* to the end of the wide-character string pointed to by *ws1*. The initial wide-character code of *ws2* shall overwrite the null wide-character code at the end of *ws1*. A terminating null wide-character code shall always be appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *wcsncat*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscat( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The *wcsncat*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcsncmp — compare part of two wide-character strings

SYNOPSIS

```
#include <wchar.h>

int wcsncmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsncmp*( ) function shall compare not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not compared) from the array pointed to by *ws1* to the array pointed to by *ws2*.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. |

RETURN VALUE

Upon successful completion, *wcsncmp*( ) shall return an integer greater than, equal to, or less than 0, if the possibly null-terminated array pointed to by *ws1* is greater than, equal to, or less than the possibly null-terminated array pointed to by *ws2*, respectively.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcsncpy — copy part of a wide-character string

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsncpy*( ) function shall copy not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not copied) from the array pointed to by *ws2* to the array pointed to by *ws1*. If copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by *ws2* is a wide-character string that is shorter than *n* wide-character codes, null wide-character codes shall be appended to the copy in the array pointed to by *ws1*, until *n* wide-character codes in all are written.

RETURN VALUE

The *wcsncpy*( ) function shall return *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

If there is no null wide-character code in the first *n* wide-character codes of the array pointed to by *ws2*, the result is not null-terminated.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscpy( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The *wcsncpy*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcspbrk — scan wide-character string for a wide-character code

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcspbrk*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcspbrk*( ) shall return a pointer to the wide-character code or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), *wcsrchr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

NAME

wcsrchr — wide-character string scanning operation

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *ws, wchar_t wc);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsrchr*( ) function shall locate the last occurrence of *wc* in the wide-character string pointed to by *ws*. The application shall ensure that the value of *wc* is a character representable as a type **wchar_t** and a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code shall be considered to be part of the wide-character string.

RETURN VALUE

Upon successful completion, *wcsrchr*( ) shall return a pointer to the wide-character code or a null pointer if *wc* does not occur in the wide-character string.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wcsrtombs — convert a wide-character string to a character string (restartable)

SYNOPSIS

```
#include <wchar.h>

size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
    size_t len, mbstate_t *restrict ps);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsrtombs*( ) function shall convert a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by *ps*. If *dst* is not a null pointer, the converted characters shall then be stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null wide character, which shall also be stored. Conversion shall stop earlier in the following cases:

- When a code is reached that does not correspond to a valid character
- When the next character would exceed the limit of *len* total bytes to be stored in the array pointed to by *dst* (and *dst* is not a null pointer)

Each conversion shall take place as if by a call to the *wcrtomb*( ) function.

If *dst* is not a null pointer, the pointer object pointed to by *src* shall be assigned either a null pointer (if conversion stopped due to reaching a terminating null wide character) or the address just past the last wide character converted (if any). If conversion stopped due to reaching a terminating null wide character, the resulting state described shall be the initial conversion state.

If *ps* is a null pointer, the *wcsrtombs*( ) function shall use its own internal **mbstate_t** object, which is initialized at program start-up to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by *ps* shall be used to completely describe the current conversion state of the associated character sequence. The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *wcsrtombs*( ).

CX If the application uses any of the _POSIX_THREAD_SAFE_FUNCTIONS or _POSIX_THREADS functions, the application shall ensure that the *wcsrtombs*( ) function is called with a non-NULL *ps* argument.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the *wcsrtombs*( ) function shall store the value of the macro [EILSEQ] in *errno* and return (**size_t**)−1; the conversion state is undefined. Otherwise, it shall return the number of bytes in the resulting character sequence, not including the terminating null (if any).

ERRORS

The *wcsrtombs*( ) function may fail if:

CX	[EINVAL]	*ps* points to an object that contains an invalid conversion state.
	[EILSEQ]	A wide-character code does not correspond to a valid character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mbsinit( ), *wcrtomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

In the DESCRIPTION, a note on using this function in a threaded application is added.

Extensions beyond the ISO C standard are now marked.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *wcsrtombs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcsspn — get length of a wide substring

SYNOPSIS

```
#include <wchar.h>

size_t wcsspn(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsspn*( ) function shall compute the length (in wide characters) of the maximum initial segment of the wide-character string pointed to by *ws1* which consists entirely of wide-character codes from the wide-character string pointed to by *ws2*.

RETURN VALUE

The *wcsspn*( ) function shall return the length of the initial substring of *ws1*; no return value is reserved to indicate an error.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscspn( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The RETURN VALUE section is updated to indicate that *wcsspn*( ) returns the length of *ws1* rather that *ws1* itself.

NAME

wcsstr — find a wide-character substring

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *restrict ws1, const wchar_t *restrict ws2);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsstr*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide characters (excluding the terminating null wide character) in the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcsstr*( ) shall return a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcschr( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The *wcsstr*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcstod, wcstof, wcstold — convert a wide-character string to a double-precision number

SYNOPSIS

```
#include <wchar.h>

double wcstod(const wchar_t *restrict nptr, wchar_t **restrict endptr);
float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
long double wcstold(const wchar_t *restrict nptr,
    wchar_t **restrict endptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they shall decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part
- One of INF or INFINITY, or any other wide string equivalent except for case
- One of NAN or NAN(*n-wchar-sequence*$_{opt}$), or any other wide string ignoring case in the NAN part, where:

```
n-wchar-sequence:
    digit
    nondigit
    n-wchar-sequence digit
    n-wchar-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the radix character (whichever occurs first) shall be interpreted as a floating constant according to the rules of the C language, except that the radix character shall be used in place of a period, and that if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero shall be assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence shall be interpreted as negated. A wide-character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, else as if it were a floating constant that is too large for the range of the return type. A wide-character sequence NAN or NAN(*n-wchar-sequence*$_{opt}$) shall be interpreted as a quiet NaN, if supported in the return type, else as if it were a subject sequence part that does not have the expected form; the meaning of the *n*-wchar sequences is implementation-defined. A pointer to the final wide string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the conversion shall be rounded in an implementation-defined manner.

CX The radix character shall be as defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character shall default to a period (`'.'`).

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *wcstod*( ) function shall not change the setting of *errno* if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstod*( ), *wcstof*( ), or *wcstold*( ), then check *errno*.

RETURN VALUE

CX Upon successful completion, these functions shall return the converted value. If no conversion could be performed, 0 shall be returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, plus or minus HUGE_VAL, HUGE_VALF, or HUGE_VALL shall be returned (according to the sign of the value), and *errno* shall be set to [ERANGE].

If the correct value would cause underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type shall be returned and *errno* set to [ERANGE].

ERRORS

The *wcstod*( ) function shall fail if:

[ERANGE] The value to be returned would cause overflow or underflow.

The *wcstod*( ) function may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

If the subject sequence has the hexadecimal form and FLT_RADIX is not a power of 2, and the result is not exactly representable, the result should be one of the two numbers in the appropriate internal format that are adjacent to the hexadecimal floating source value, with the extra stipulation that the error should have a correct sign for the current rounding direction.

If the subject sequence has the decimal form and at most DECIMAL_DIG (defined in **<float.h>**) significant digits, the result should be correctly rounded. If the subject sequence *D* has the decimal form and more than DECIMAL_DIG significant digits, consider the two bounding, adjacent decimal strings *L* and *U*, both having DECIMAL_DIG significant digits, such that the values of *L*, *D*, and *U* satisfy `"L <= D <= U"`. The result should be one of the (equal or adjacent) values that would be obtained by correctly rounding *L* and *U* according to the current rounding direction, with the extra stipulation that the error with respect to *D* should have a correct sign for the current rounding direction.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswspace( ), *localeconv*( ), *scanf*( ), *setlocale*( ), *wcstol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<float.h>**, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO/IEC 9899:1999 standard:

- The *wcstod*( ) prototype is updated.
- The *wcstof*( ) and *wcstold*( ) functions are added.
- If the correct value for *wcstod*( ) would cause underflow, the return value changed from 0 (as specified in Issue 5) to the smallest normalized positive number.
- The DESCRIPTION, RETURN VALUE, and APPLICATION USAGE sections are extensively updated.

ISO/IEC 9899:1999 standard, Technical Corrigendum No. 1 is incorporated.

NAME

wcstof — convert a wide-character string to a double-precision number

SYNOPSIS

```
#include <wchar.h>

float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
```

DESCRIPTION

Refer to *wcstod*( ).

NAME

wcstoimax, wcstoumax — convert wide-character string to integer type

SYNOPSIS

```
#include <stddef.h>
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall be equivalent to the *wcstol*( ), *wcstoll*( ), *wcstoul*( ), and *wcstoull*( ) functions, respectively, except that the initial portion of the wide string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

RETURN VALUE

These functions shall return the converted value, if any.

If no conversion could be performed, zero shall be returned. If the correct value is outside the range of representable values, {INTMAX_MAX}, {INTMAX_MIN}, or {UINTMAX_MAX} shall be returned (according to the return type and sign of the value, if any), and *errno* shall be set to [ERANGE].

ERRORS

These functions shall fail if:

[EINVAL]	The value of *base* is not supported.
[ERANGE]	The value to be returned is not representable.

These functions may fail if:

[EINVAL]	No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcstol( ), *wcstoul*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<inttypes.h>**, **<stddef.h>**

CHANGE HISTORY

First released in Issue 6. Derived from the ISO / IEC 9899: 1999 standard.

NAME

wcstok — split wide-character string into tokens

SYNOPSIS

```
#include <wchar.h>

wchar_t *wcstok(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    wchar_t **restrict ptr);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

A sequence of calls to *wcstok*( ) shall break the wide-character string pointed to by *ws1* into a sequence of tokens, each of which shall be delimited by a wide-character code from the wide-character string pointed to by *ws2*. The *ptr* argument points to a caller-provided **wchar_t** pointer into which the *wcstok*( ) function shall store information necessary for it to continue scanning the same wide-character string. |

The first call in the sequence has *ws1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *ws2* may be different from call to call.

The first call in the sequence shall search the wide-character string pointed to by *ws1* for the first wide-character code that is *not* contained in the current separator string pointed to by *ws2*. If no such wide-character code is found, then there are no tokens in the wide-character string pointed to by *ws1* and *wcstok*( ) shall return a null pointer. If such a wide-character code is found, it shall be the start of the first token. |

The *wcstok*( ) function shall then search from there for a wide-character code that *is* contained in the current separator string. If no such wide-character code is found, the current token extends to the end of the wide-character string pointed to by *ws1*, and subsequent searches for a token shall return a null pointer. If such a wide-character code is found, it shall be overwritten by a null wide character, which terminates the current token. The *wcstok*( ) function shall save a pointer to the following wide-character code, from which the next search for a token shall start. |

Each subsequent call, with a null pointer as the value of the first argument, shall start searching from the saved pointer and behave as described above. |

The implementation shall behave as if no function calls *wcstok*( ).

RETURN VALUE

Upon successful completion, the *wcstok*( ) function shall return a pointer to the first wide-character code of a token. Otherwise, if there is no token, *wcstok*( ) shall return a null pointer.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

Aligned with ISO/IEC 9899:1990/Amendment 1:1995 (E). Specifically, a third argument is added to the definition of this function in the SYNOPSIS.

Issue 6

The *wcstok*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcstol, wcstoll — convert a wide-character string to a long integer

SYNOPSIS

```
#include <wchar.h>

long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
    int base);
long long wcstoll(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

These functions shall convert the initial portion of the wide-character string pointed to by *nptr* to **long**, **long long**, **unsigned long**, and **unsigned long long** representation, respectively. First, they shall decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to an integer, and return the result.

If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a `'+'` or `'−'` sign, but not including an integer suffix. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* shall be permitted. If the value of *base* is 16, the wide-character code representations of 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white-space wide-character code that is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX These functions shall not change the setting of *errno* if successful.

Since 0, {LONG_MIN} or {LLONG_MIN} and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstol*( ) or *wcstoll*( ), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned CX and *errno* may be set to indicate the error. If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX}, {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [EINVAL]	The value of *base* is not supported.
[ERANGE]	The value to be returned is not representable.

These functions may fail if:

CX [EINVAL]	No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha( ), *scanf*( ), *wcstod*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *wcstol*( ) prototype is updated.
- The *wcstoll*( ) function is added.

NAME

wcstold — convert a wide-character string to a double-precision number

SYNOPSIS

```
#include <wchar.h>

long double wcstold(const wchar_t *restrict nptr,
    wchar_t **restrict endptr);
```

DESCRIPTION

Refer to *wcstod*( ).

NAME

wcstoll — convert a wide-character string to a long integer

SYNOPSIS

```
#include <wchar.h>

long long wcstoll(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

Refer to *wcstol*( ).

NAME

wcstombs — convert a wide-character string to a character string

SYNOPSIS

```
#include <stdlib.h>

size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcstombs*( ) function shall convert the sequence of wide-character codes that are in the array pointed to by *pwcs* into a sequence of characters that begins in the initial shift state and store these characters into the array pointed to by *s*, stopping if a character would exceed the limit of *n* total bytes or if a null byte is stored. Each wide-character code shall be converted as if by a call to *wctomb*( ), except that the shift state of *wctomb*( ) shall not be affected.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

No more than *n* bytes shall be modified in the array pointed to by *s*. If copying takes place between objects that overlap, the behavior is undefined. CX If *s* is a null pointer, *wcstombs*( ) shall return the length required to convert the entire array regardless of the value of *n*, but no values are stored.

The *wcstombs*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

RETURN VALUE

If a wide-character code is encountered that does not correspond to a valid character (of one or more bytes each), *wcstombs*( ) shall return (**size_t**)−1. Otherwise, *wcstombs*( ) shall return the number of bytes stored in the character array, not including any terminating null byte. The array shall not be null-terminated if the value returned is *n*.

ERRORS

The *wcstombs*( ) function may fail if:

CX [EILSEQ] A wide-character code does not correspond to a valid character.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mblen( ), *mbtowc*( ), *mbstowcs*( ), *wctomb*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ISO C standard.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION states the effect of when *s* is a null pointer.
- The [EILSEQ] error condition is added.

The *wcstombs*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wcstoul, wcstoull — convert a wide-character string to an unsigned long

SYNOPSIS

```
#include <wchar.h>

unsigned long wcstoul(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
unsigned long long wcstoull(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcstoul*( ) and *wcstoull*( ) functions shall convert the initial portion of the wide-character string pointed to by *nptr* to **unsigned long** and **unsigned long long** representation, respectively. First, they shall decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by *iswspace*( ))
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then they shall attempt to convert the subject sequence to an unsigned integer, and return the result.

If *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a `'+'` or `'−'` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `'0'` optionally followed by a sequence of the digits `'0'` to `'7'` only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters `'a'` (or `'A'`) to `'f'` (or `'F'`) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a `'+'` or `'−'` sign, but not including an integer suffix. The letters from `'a'` (or `'A'`) to `'z'` (or `'Z'`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* shall be permitted. If the value of *base* is 16, the wide-character codes 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not white space and is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character codes, or if the first wide-character code that is not white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and *base* is 0, the sequence of wide-character codes starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion shall be negated. A pointer to the final wide-character string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

CX The *wcstoul*( ) function shall not change the setting of *errno* if successful.

Since 0, {ULONG_MAX}, and {ULLONG_MAX} are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *wcstoul*( ) or *wcstoull*( ), then check *errno*.

RETURN VALUE

Upon successful completion, the *wcstoul*( ) and *wcstoull*( ) functions shall return the converted CX value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to indicate the error. If the correct value is outside the range of representable values, {ULONG_MAX} or {ULLONG_MAX} respectively shall be returned and *errno* set to [ERANGE].

ERRORS

These functions shall fail if:

CX [EINVAL] The value of *base* is not supported.

[ERANGE] The value to be returned is not representable.

These functions may fail if:

CX [EINVAL] No conversion could be performed.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswalpha( ), *scanf*( ), *wcstod*( ), *wcstol*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The [EINVAL] error condition is added for when the value of *base* is not supported.

In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The following changes are made for alignment with the ISO / IEC 9899: 1999 standard:

- The *wcstoul*( ) prototype is updated.
- The *wcstoull*( ) function is added.

NAME

wcstoull — convert a wide-character string to an unsigned long

SYNOPSIS

```
#include <wchar.h>

unsigned long long wcstoull(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

Refer to *wcstoul*( ).

NAME

wcstoumax — convert wide-character string to integer type

SYNOPSIS

```
#include <stddef.h>
#include <inttypes.h>

uintmax_t wcstoumax(const wchar_t *restrict nptr,
    wchar_t **restrict endptr, int base);
```

DESCRIPTION

Refer to *wcstoimax*( ).

NAME

wcswcs — find a wide substring (**LEGACY**)

SYNOPSIS

XSI

```
#include <wchar.h>

wchar_t *wcswcs(const wchar_t *ws1, const wchar_t *ws2);
```

DESCRIPTION

The *wcswcs*( ) function shall locate the first occurrence in the wide-character string pointed to by *ws1* of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide-character string pointed to by *ws2*.

RETURN VALUE

Upon successful completion, *wcswcs*( ) shall return a pointer to the located wide-character string or a null pointer if the wide-character string is not found.

If *ws2* points to a wide-character string with zero length, the function shall return *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function was not included in the final ISO/IEC 9899:1990/Amendment 1:1995 (E). Application developers are strongly encouraged to use the *wcsstr*( ) function instead.

RATIONALE

None.

FUTURE DIRECTIONS

This function may be withdrawn in a future version.

SEE ALSO

wcschr( ), *wcsstr*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Marked EX.

Issue 6

This function is marked LEGACY.

NAME

wcswidth — number of column positions of a wide-character string

SYNOPSIS

XSI

```
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);
```

DESCRIPTION

The *wcswidth*( ) function shall determine the number of column positions required for *n* wide-character codes (or fewer than *n* wide-character codes if a null wide-character code is encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

RETURN VALUE

The *wcswidth*( ) function either shall return 0 (if *pwcs* points to a null wide-character code), or return the number of column positions to be occupied by the wide-character string pointed to by *pwcs*, or return –1 (if any of the first *n* wide-character codes in the wide-character string pointed to by *pwcs* is not a printable wide-character code).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcwidth( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**, the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.103, Column Position

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 6

The Open Group Corrigendum U021/11 is applied. The function is marked as an extension.

NAME

wcsxfrm — wide-character string transformation

SYNOPSIS

```
#include <wchar.h>

size_t wcsxfrm(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wcsxfrm*( ) function shall transform the wide-character string pointed to by *ws2* and place the resulting wide-character string into the array pointed to by *ws1*. The transformation shall be such that if *wcscmp*( ) is applied to two transformed wide strings, it shall return a value greater than, equal to, or less than 0, corresponding to the result of *wcscoll*( ) applied to the same two original wide-character strings. No more than *n* wide-character codes shall be placed into the resulting array pointed to by *ws1*, including the terminating null wide-character code. If *n* is 0, *ws1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

CX The *wcsxfrm*( ) function shall not change the setting of *errno* if successful.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *wcsxfrm*( ), then check *errno*.

RETURN VALUE

The *wcsxfrm*( ) function shall return the length of the transformed wide-character string (not including the terminating null wide-character code). If the value returned is *n* or more, the contents of the array pointed to by *ws1* are unspecified.

CX On error, the *wcsxfrm*( ) function may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *wcsxfrm*( ) function may fail if:

CX [EINVAL] The wide-character string pointed to by *ws2* contains wide-character codes outside the domain of the collating sequence.

EXAMPLES

None.

APPLICATION USAGE

The transformation function is such that two transformed wide-character strings can be ordered by *wcscmp*( ) as appropriate to collating sequence information in the program's locale (category *LC_COLLATE*).

The fact that when *n* is 0 *ws1* is permitted to be a null pointer is useful to determine the size of the *ws1* array prior to making the transformation.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcscmp( ), *wcscoll*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the MSE working draft.

Issue 5

Moved from ENHANCED I18N to BASE and the [ENOSYS] error is removed.

The DESCRIPTION is updated to indicate that *errno* is not changed if the function is successful.

Issue 6

In previous versions, this function was required to return –1 on error.

Extensions beyond the ISO C standard are now marked.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the RETURN VALUE and ERRORS sections, the [EINVAL] optional error condition is added if no conversion could be performed.

The *wcsxfrm*( ) prototype is updated for alignment with the ISO / IEC 9899: 1999 standard.

NAME

wctob — wide-character to single-byte conversion

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int wctob(wint_t c);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wctob*( ) function shall determine whether *c* corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

The behavior of this function shall be affected by the *LC_CTYPE* category of the current locale.

RETURN VALUE

The *wctob*( ) function shall return EOF if *c* does not correspond to a character with length one in the initial shift state. Otherwise, it shall return the single-byte representation of that character as an **unsigned char** converted to **int**.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

btowc( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

NAME

wctomb — convert a wide-character code to a character

SYNOPSIS

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wctomb*( ) function shall determine the number of bytes needed to represent the character corresponding to the wide-character code whose value is *wchar* (including any change in the shift state). It shall store the character representation (possibly multiple bytes and any special bytes to change shift state) in the array object pointed to by *s* (if *s* is not a null pointer). At most {MB_CUR_MAX} bytes shall be stored. If *wchar* is 0, a null byte shall be stored, preceded by any shift sequence needed to restore the initial shift state, and *wctomb*( ) shall be left in the initial shift state.

CX The behavior of this function is affected by the *LC_CTYPE* category of the current locale. For a state-dependent encoding, this function shall be placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer shall cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer shall cause this function to return a non-zero value if encodings have state dependency, and 0 otherwise. Changing the *LC_CTYPE* category causes the shift state of this function to be unspecified.

The *wctomb*( ) function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

The implementation shall behave as if no function defined in this volume of IEEE Std 1003.1-200x calls *wctomb*( ).

RETURN VALUE

If *s* is a null pointer, *wctomb*( ) shall return a non-zero or 0 value, if character encodings, respectively, do or do not have state-dependent encodings. If *s* is not a null pointer, *wctomb*( ) shall return −1 if the value of *wchar* does not correspond to a valid character, or return the number of bytes that constitute the character corresponding to the value of *wchar*.

In no case shall the value returned be greater than the value of the {MB_CUR_MAX} macro.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mblen( ), *mbtowc*( ), *mbstowcs*( ), *wcstombs*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<stdlib.h>**

CHANGE HISTORY

First released in Issue 4. Derived from the ANSI C standard.

Issue 6

Extensions beyond the ISO C standard are now marked.

In the DESCRIPTION, a note about reentrancy and thread-safety is added.

NAME

wctrans — define character mapping

SYNOPSIS

```
#include <wctype.h>

wctrans_t wctrans(const char *charclass);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wctrans*( ) function is defined for valid character mapping names identified in the current locale. The *charclass* is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales: **tolower** and **toupper**.

The function shall return a value of type **wctrans_t**, which can be used as the second argument to subsequent calls of *towctrans*( ). The *wctrans*( ) function shall determine values of **wctrans_t** according to the rules of the coded character set defined by character mapping information in the program's locale (category *LC_CTYPE*). The values returned by *wctrans*( ) shall be valid until a call to *setlocale*( ) that modifies the category *LC_CTYPE*.

RETURN VALUE

CX The *wctrans*( ) function shall return 0 and may set *errno* to indicate the error if the given character mapping name is not valid for the current locale (category *LC_CTYPE*); otherwise, it shall return a non-zero object of type **wctrans_t** that can be used in calls to *towctrans*( ).

ERRORS

The *wctrans*( ) function may fail if:

CX [EINVAL] The character mapping name pointed to by *charclass* is not valid in the current locale.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

towctrans( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**

CHANGE HISTORY

First released in Issue 5. Derived from ISO/IEC 9899:1990/Amendment 1:1995 (E).

NAME

wctype — define character class

SYNOPSIS

```
#include <wctype.h>

wctype_t wctype(const char *property);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wctype*( ) function is defined for valid character class names as defined in the current locale. The *property* argument is a string identifying a generic character class for which codeset-specific type information is required. The following character class names shall be defined in all locales:

alnum	**digit**	**punct**
alpha	**graph**	**space**
blank	**lower**	**upper**
cntrl	**print**	**xdigit**

Additional character class names defined in the locale definition file (category *LC_CTYPE*) can also be specified.

The function shall return a value of type **wctype_t**, which can be used as the second argument to subsequent calls of *iswctype*( ). The *wctype*( ) function shall determine values of **wctype_t** according to the rules of the coded character set defined by character type information in the program's locale (category *LC_CTYPE*). The values returned by *wctype*( ) shall be valid until a call to *setlocale*( ) that modifies the category *LC_CTYPE*.

RETURN VALUE

The *wctype*( ) function shall return 0 if the given character class name is not valid for the current locale (category *LC_CTYPE*); otherwise, it shall return an object of type **wctype_t** that can be used in calls to *iswctype*( ).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iswctype( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wctype.h>**, **<wchar.h>**

CHANGE HISTORY

First released in Issue 4.

Issue 5

The following change has been made in this issue for alignment with ISO / IEC 9899: 1990 / Amendment 1: 1995 (E):

- The SYNOPSIS has been changed to indicate that this function and associated data types are now made visible by inclusion of the **<wctype.h>** header rather than **<wchar.h>**.

NAME

wcwidth — number of column positions of a wide-character code

SYNOPSIS

XSI
```
#include <wchar.h>

int wcwidth(wchar_t wc);
```

DESCRIPTION

The *wcwidth*( ) function shall determine the number of column positions required for the wide character *wc*. The application shall ensure that the value of *wc* is a character representable as a **wchar_t**, and is a wide-character code corresponding to a valid character in the current locale.

RETURN VALUE

The *wcwidth*( ) function shall either return 0 (if *wc* is a null wide-character code), or return the number of column positions to be occupied by the wide-character code *wc*, or return −1 (if *wc* does not correspond to a printable wide-character code).

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

This function was removed from the final ISO/IEC 9899:1990/Amendment 1:1995 (E), and the return value for a non-printable wide character is not specified.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wcswidth( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released as a World-wide Portability Interface in Issue 4. Derived from MSE working draft.

Issue 6

The Open Group Corrigendum U021/12 is applied. This function is marked as an extension.

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wmemchr — find a wide character in memory

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wmemchr*( ) function shall locate the first occurrence of *wc* in the initial *n* wide characters of the object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially. |

If *n* is zero, the application shall ensure that *ws* is a valid pointer and the function behaves as if no valid occurrence of *wc* is found.

RETURN VALUE

The *wmemchr*( ) function shall return a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemcmp( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899: 1990/Amendment 1: 1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ''must'' for application requirements.

NAME

wmemcmp — compare wide characters in memory

SYNOPSIS

```
#include <wchar.h>

int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wmemcmp*( ) function shall compare the first *n* wide characters of the object pointed to by *ws1* to the first *n* wide characters of the object pointed to by *ws2*. This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially. |

If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function shall behave as if the two objects compare equal. |

RETURN VALUE

The *wmemcmp*( ) function shall return an integer greater than, equal to, or less than zero, respectively, as the object pointed to by *ws1* is greater than, equal to, or less than the object pointed to by *ws2*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr( ), *wmemcpy*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wmemcpy — copy wide characters in memory

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t *restrict ws1, const wchar_t *restrict ws2,
    size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wmemcpy*( ) function shall copy *n* wide characters from the object pointed to by *ws2* to the object pointed to by *ws1*. This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially. |

If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function shall copy zero wide characters. |

RETURN VALUE

The *wmemcpy*( ) function shall return the value of *ws1*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr( ), *wmemcmp*( ), *wmemmove*( ), *wmemset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The *wmemcpy*( ) prototype is updated for alignment with the ISO/IEC 9899:1999 standard.

NAME

wmemmove — copy wide characters in memory with overlapping areas

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wmemmove*( ) function shall copy *n* wide characters from the object pointed to by *ws2* to the object pointed to by *ws1*. Copying shall take place as if the *n* wide characters from the object pointed to by *ws2* are first copied into a temporary array of *n* wide characters that does not overlap the objects pointed to by *ws1* or *ws2*, and then the *n* wide characters from the temporary array are copied into the object pointed to by *ws1*. |

This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially. |

If *n* is zero, the application shall ensure that *ws1* and *ws2* are valid pointers, and the function shall copy zero wide characters. |

RETURN VALUE

The *wmemmove*( ) function shall return the value of *ws1*.

ERRORS

No errors are defined

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr( ), *wmemcmp*( ), *wmemcpy*( ), *wmemset*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wmemset — set wide characters in memory

SYNOPSIS

```
#include <wchar.h>

wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);
```

DESCRIPTION

CX The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.

The *wmemset*( ) function shall copy the value of *wc* into each of the first *n* wide characters of the object pointed to by *ws*. This function shall not be affected by locale and all **wchar_t** values shall be treated identically. The null wide character and **wchar_t** values not corresponding to valid characters shall not be treated specially.

If *n* is zero, the application shall ensure that *ws* is a valid pointer, and the function shall copy zero wide characters.

RETURN VALUE

The *wmemset*( ) functions shall return the value of *ws*.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

wmemchr( ), *wmemcmp*( ), *wmemcpy*( ), *wmemmove*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wchar.h>**

CHANGE HISTORY

First released in Issue 5. Included for alignment with ISO/IEC 9899:1990/Amendment 1:1995 (E).

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

NAME

wordexp, wordfree — perform word expansions

SYNOPSIS

```
#include <wordexp.h>

int wordexp(const char *restrict words, wordexp_t *restrict pwordexp,
    int flags);
void wordfree(wordexp_t *pwordexp);
```

DESCRIPTION

The *wordexp*( ) function shall perform word expansions as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions, subject to quoting as in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.2, Quoting, and place the list of expanded words into the structure pointed to by *pwordexp*.

The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions shall be the same as would be performed by the command line interpreter if *words* were the part of a command line representing the arguments to a utility. Therefore, the application shall ensure that *words* does not contain an unquoted <newline> or any of the unquoted shell special characters '|', '&', ';', '<', '>' except in the context of command substitution as specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3, Command Substitution. It also shall not contain unquoted parentheses or braces, except in the context of command or variable substitution. The application shall ensure that every member of *words* which it expects to have expanded by *wordexp*( ) does not contain an unquoted initial comment character. The application shall also ensure that any words which it intends to be ignored (because they begin or continue a comment) are deleted from *words*. If the argument *words* contains an unquoted comment character (number sign) that is the beginning of a token, *wordexp*( ) shall either treat the comment character as a regular character, or interpret it as a comment indicator and ignore the remainder of *words*.

The structure type **wordexp_t** is defined in the **<wordexp.h>** header and includes at least the following members:

Member Type	Member Name	Description
size_t	*we_wordc*	Count of words matched by *words*.
char **	*we_wordv*	Pointer to list of expanded words.
size_t	*we_offs*	Slots to reserve at the beginning of *pwordexp*->*we_wordv*.

The *wordexp*( ) function shall store the number of generated words into *pwordexp*->*we_wordc* and a pointer to a list of pointers to words in *pwordexp*->*we_wordv*. Each individual field created during field splitting (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.5, Field Splitting) or pathname expansion (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.6, Pathname Expansion) shall be a separate word in the *pwordexp*->*we_wordv* list. The words shall be in order as described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6, Word Expansions. The first pointer after the last word pointer shall be a null pointer. The expansion of special parameters described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.5.2, Special Parameters is unspecified.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The *wordexp*( ) function shall allocate other space as needed, including memory pointed to by *pwordexp*->*we_wordv*. The *wordfree*( ) function frees any memory associated with *pwordexp* from a previous call to *wordexp*( ).

The *flags* argument is used to control the behavior of *wordexp*( ). The value of *flags* is the bitwise-inclusive OR of zero or more of the following constants, which are defined in **<wordexp.h>**:

WRDE_APPEND	Append words generated to the ones from a previous call to *wordexp*( ).
WRDE_DOOFFS	Make use of *pwordexp->we_offs*. If this flag is set, *pwordexp->we_offs* is used to specify how many null pointers to add to the beginning of *pwordexp->we_wordv*. In other words, *pwordexp->we_wordv* shall point to *pwordexp->we_offs* null pointers, followed by *pwordexp->we_wordc* word pointers, followed by a null pointer.
WRDE_NOCMD	If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, fail if command substitution, as specified in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.3, Command Substitution, is requested.
WRDE_REUSE	The *pwordexp* argument was passed to a previous successful call to *wordexp*( ), and has not been passed to *wordfree*( ). The result shall be the same as if the application had called *wordfree*( ) and then called *wordexp*( ) without WRDE_REUSE.
WRDE_SHOWERR	Do not redirect *stderr* to **/dev/null**.
WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to *wordexp*( ). The following rules apply to applications when two or more calls to *wordexp*( ) are made with the same value of *pwordexp* and without intervening calls to *wordfree*( ):

1. The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.
2. All of the calls shall set WRDE_DOOFFS, or all shall not set it.
3. After the second and each subsequent call, *pwordexp->we_wordv* shall point to a list containing the following:
   a. Zero or more null pointers, as specified by WRDE_DOOFFS and *pwordexp->we_offs*
   b. Pointers to the words that were in the *pwordexp->we_wordv* list before the call, in the same order as before
   c. Pointers to the new words generated by the latest call, in the specified order
4. The count returned in *pwordexp->we_wordc* shall be the total number of words from all of the calls.
5. The application can change any of the fields after a call to *wordexp*( ), but if it does it shall reset them to the original value before a subsequent call, using the same *pwordexp* value, to *wordfree*( ) or *wordexp*( ) with the WRDE_APPEND or WRDE_REUSE flag.

If the implementation supports the utilities defined in the Shell and Utilities volume of IEEE Std 1003.1-200x, and *words* contains an unquoted character—<newline>, '|', '&', ';', '<', '>', '(', ')', '{', '}'—in an inappropriate context, *wordexp*( ) shall fail, and the number of expanded words shall be 0.

Unless WRDE_SHOWERR is set in *flags*, *wordexp*( ) shall redirect *stderr* to **/dev/null** for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERR is set, *wordexp*( ) may write messages to *stderr* if syntax errors are detected while expanding *words*.

The application shall ensure that if WRDE_DOOFFS is set, then *pwordexp*->*we_offs* has the same value for each *wordexp*( ) call and *wordfree*( ) call using a given *pwordexp*.

The following constants are defined as error return values:

WRDE_BADCHAR	One of the unquoted characters—<newline>, '|', '&', ';', '<', '>', '(', ')', '{', '}'—appears in *words* in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.
WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in flags.
WRDE_NOSPACE	Attempt to allocate memory failed.
WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

RETURN VALUE

Upon successful completion, *wordexp*( ) shall return 0. Otherwise, a non-zero value, as described in **<wordexp.h>**, shall be returned to indicate an error. If *wordexp*( ) returns the value WRDE_NOSPACE, then *pwordexp*->*we_wordc* and *pwordexp*->*we_wordv* shall be updated to reflect any words that were successfully expanded. In other cases, they shall not be modified.

The *wordfree*( ) function shall not return a value.

ERRORS

No errors are defined.

EXAMPLES

None.

APPLICATION USAGE

The *wordexp*( ) function is intended to be used by an application that wants to do all of the shell's expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or list of filenames) and then uses *wordexp*( ) to process the input, the user could respond with anything that would be valid as input to the shell.

The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects, such as executing a command or writing a file.

RATIONALE

This function was included as an alternative to *glob*( ). There had been continuing controversy over exactly what features should be included in *glob*( ). It is hoped that by providing *wordexp*( ) (which provides all of the shell word expansions, but which may be slow to execute) and *glob*( ) (which is faster, but which only performs pathname expansion, without tilde or parameter expansion) this will satisfy the majority of applications.

While *wordexp*( ) could be implemented entirely as a library routine, it is expected that most implementations run a shell in a subprocess to do the expansion.

Two different approaches have been proposed for how the required information might be presented to the shell and the results returned. They are presented here as examples.

One proposal is to extend the *echo* utility by adding a **−q** option. This option would cause *echo* to add a backslash before each backslash and <blank> that occurs within an argument. The *wordexp*( ) function could then invoke the shell as follows:

```
(void) strcpy(buffer, "echo -q");
(void) strcat(buffer, words);
if ((flags & WRDE_SHOWERR) == 0)
```

```
        (void) strcat(buffer, "2>/dev/null");
    f = popen(buffer, "r");
```

The *wordexp*( ) function would read the resulting output, remove unquoted backslashes, and break into words at unquoted <blank>s. If the WRDE_NOCMD flag was set, *wordexp*( ) would have to scan *words* before starting the subshell to make sure that there would be no command substitution. In any case, it would have to scan *words* for unquoted special characters.

Another proposal is to add the following options to *sh*:

−w *wordlist*
This option provides a wordlist expansion service to applications. The words in *wordlist* shall be expanded and the following written to standard output:

1. The count of the number of words after expansion, in decimal, followed by a null byte
2. The number of bytes needed to represent the expanded words (not including null separators), in decimal, followed by a null byte
3. The expanded words, each terminated by a null byte

If an error is encountered during word expansion, *sh* exits with a non-zero status after writing the former to report any words successfully expanded

−P Run in ‘‘protected’’ mode. If specified with the −**w** option, no command substitution shall be performed.

With these options, *wordexp*( ) could be implemented fairly simply by creating a subprocess using *fork*( ) and executing *sh* using the line:

```
execl(<shell path>, "sh", "-P", "-w", words, (char *)0);
```

after directing standard error to **/dev/null**.

It seemed objectionable for a library routine to write messages to standard error, unless explicitly requested, so *wordexp*( ) is required to redirect standard error to **/dev/null** to ensure that no messages are generated, even for commands executed for command substitution. The WRDE_SHOWERR flag can be specified to request that error messages be written.

The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and reallocating memory, if that is possible. A minimal implementation can call *wordfree*( ) when WRDE_REUSE is set.

FUTURE DIRECTIONS

None.

SEE ALSO

fnmatch( ), *glob*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<wordexp.h>**, the Shell and Utilities volume of IEEE Std 1003.1-200x

CHANGE HISTORY

First released in Issue 4. Derived from the ISO POSIX-2 standard.

Issue 5

Moved from POSIX2 C-language Binding to BASE.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The **restrict** keyword is added to the *wordexp*( ) prototype for alignment with the ISO/IEC 9899:1999 standard.

NAME

wprintf — print formatted wide-character output

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int wprintf(const wchar_t *restrict format, ...);
```

DESCRIPTION

Refer to *fwprintf*( ).

NAME

pwrite, write — write on a file

SYNOPSIS

```
#include <unistd.h>

XSI ssize_t pwrite(int fildes, const void *buf, size_t nbyte,
        off_t offset);
    ssize_t write(int fildes, const void *buf, size_t nbyte);
```

DESCRIPTION

The *write*( ) function shall attempt to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.

Before any action described below is taken, and if *nbyte* is zero and the file is a regular file, the *write*( ) function may detect and return errors as described below. In the absence of errors, or if error detection is not performed, the *write*( ) function shall return zero and have no other results. If *nbyte* is zero and the file is not a regular file, the results are unspecified.

On a regular file or other file capable of seeking, the actual writing of data shall proceed from the position in the file indicated by the file offset associated with *fildes*. Before successful return from *write*( ), the file offset shall be incremented by the number of bytes actually written. On a regular file, if this incremented file offset is greater than the length of the file, the length of the file shall be set to this file offset.

On a file not capable of seeking, writing shall always take place starting at the current position. The value of a file offset associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file offset shall be set to the end of the file prior to each write and no intervening file modification operation shall occur between changing the file offset and the write operation.

XSI If a *write*( ) requests that more bytes be written than there is room for (for example, the process' file size limit or the physical end of a medium), only as many bytes as there is room for shall be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a non-zero number of bytes would give a failure return (except as noted below).

XSI If the request would cause the file size to exceed the soft file size limit for the process and there is no room for any bytes to be written, the request shall fail and the implementation shall generate the SIGXFSZ signal for the thread.

If *write*( ) is interrupted by a signal before it writes any data, it shall return −1 with *errno* set to [EINTR].

If *write*( ) is interrupted by a signal after it successfully writes some data, it shall return the number of bytes written.

If the value of *nbyte* is greater than {SSIZE_MAX}, the result is implementation-defined.

After a *write*( ) to a regular file has successfully returned:

- Any successful *read*( ) from each byte position in the file that was modified by that write shall return the data specified by the *write*( ) for that position until such byte positions are again modified.

- Any subsequent successful *write*( ) to the same byte position in the file shall overwrite that file data.

Write requests to a pipe or FIFO shall be handled in the same way as a regular file with the following exceptions:

- There is no file offset associated with a pipe, hence each write request shall append to the end of the pipe.
- Write requests of {PIPE_BUF} bytes or less shall not be interleaved with data from other processes doing writes on the same pipe. Writes of greater than {PIPE_BUF} bytes may have data interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag of the file status flags is set.
- If the O_NONBLOCK flag is clear, a write request may cause the thread to block, but on normal completion it shall return *nbyte*.
- If the O_NONBLOCK flag is set, *write*( ) requests shall be handled differently, in the following ways:
  - — The *write*( ) function shall not block the thread.
  - — A write request for {PIPE_BUF} or fewer bytes shall have the following effect: if there is sufficient space available in the pipe, *write*( ) shall transfer all the data and return the number of bytes requested. Otherwise, *write*( ) shall transfer no data and return –1 with *errno* set to [EAGAIN].
  - — A write request for more than {PIPE_BUF} bytes shall cause one of the following:
    - — When at least one byte can be written, transfer what it can and return the number of bytes written. When all data previously written to the pipe is read, it shall transfer at least {PIPE_BUF} bytes.
    - — When no data can be written, transfer no data, and return –1 with *errno* set to [EAGAIN].

When attempting to write to a file descriptor (other than a pipe or FIFO) that supports non-blocking writes and cannot accept the data immediately:

- If the O_NONBLOCK flag is clear, *write*( ) shall block the calling thread until the data can be accepted.
- If the O_NONBLOCK flag is set, *write*( ) shall not block the thread. If some data can be written without blocking the thread, *write*( ) shall write what it can and return the number of bytes written. Otherwise, it shall return –1 and set *errno* to [EAGAIN].

Upon successful completion, where *nbyte* is greater than 0, *write*( ) shall mark for update the *st_ctime* and *st_mtime* fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID bits of the file mode may be cleared.

For regular files, no data transfer shall occur past the offset maximum established in the open file description associated with *fildes*.

If *fildes* refers to a socket, *write*( ) shall be equivalent to *send*( ) with no flags set.

SIO If the O_DSYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O data integrity completion.

If the O_SYNC bit has been set, write I/O operations on the file descriptor shall complete as defined by synchronized I/O file integrity completion.

SHM If *fildes* refers to a shared memory object, the result of the *write*( ) function is unspecified.

TYM If *fildes* refers to a typed memory object, the result of the *write*( ) function is unspecified.

XSR If *fildes* refers to a STREAM, the operation of *write*( ) shall be determined by the values of the minimum and maximum *nbyte* range (packet size) accepted by the STREAM. These values are determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte* bytes shall be written. If *nbyte* does not fall within the range and the minimum packet size value is 0, *write*( ) shall break the buffer into maximum packet size segments prior to sending the data downstream (the last segment may contain less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, *write*( ) shall fail with *errno* set to [ERANGE]. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and 0 is returned. The process may issue I_SWROPT *ioctl*( ) to enable zero-length messages to be sent across the pipe or FIFO.

When writing to a STREAM, data messages are created with a priority band of 0. When writing to a STREAM that is not a pipe or FIFO:

- If O_NONBLOCK is clear, and the STREAM cannot accept data (the STREAM write queue is full due to internal flow control conditions), *write*( ) shall block until data can be accepted.
- If O_NONBLOCK is set and the STREAM cannot accept data, *write*( ) shall return −1 and set *errno* to [EAGAIN].
- If O_NONBLOCK is set and part of the buffer has been written while a condition in which the STREAM cannot accept additional data occurs, *write*( ) shall terminate and return the number of bytes written.

In addition, *write*( ) shall fail if the STREAM head has processed an asynchronous error before the call. In this case, the value of *errno* does not reflect the result of *write*( ), but reflects the prior error.

XSI The *pwrite*( ) function shall be equivalent to *write*( ), except that it writes into a given position without changing the file pointer. The first three arguments to *pwrite*( ) are the same as *write*( ) with the addition of a fourth argument offset for the desired position inside the file.

RETURN VALUE

XSI Upon successful completion, *write*( ) and *pwrite*( ) shall return the number of bytes actually written to the file associated with *fildes*. This number shall never be greater than *nbyte*. Otherwise, −1 shall be returned and *errno* set to indicate the error.

ERRORS

XSI The *write*( ) and *pwrite*( ) functions shall fail if:

[EAGAIN]	The O_NONBLOCK flag is set for the file descriptor and the thread would be delayed in the *write*( ) operation.
[EBADF]	The *fildes* argument is not a valid file descriptor open for writing.
[EFBIG]	An attempt was made to write a file that exceeds the implementation-defined maximum file size XSI or the process' file size limit, and there was no room for any bytes to be written.
[EFBIG]	The file is a regular file, *nbyte* is greater than 0, and the starting position is greater than or equal to the offset maximum established in the open file description associated with *fildes*.
[EINTR]	The write operation was terminated due to the receipt of a signal, and no data was transferred.
[EIO]	The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor

	blocking SIGTTOU, and the process group of the process is orphaned. This error may also be returned under implementation-defined conditions.
[ENOSPC]	There was no free space remaining on the device containing the file.
[EPIPE]	An attempt is made to write to a pipe or FIFO that is not open for reading by any process, or that only has one end open. A SIGPIPE signal shall also be sent to the thread.
XSR [ERANGE]	The transfer request size was outside the range supported by the STREAMS file associated with *fildes*.

The *write*( ) function shall fail if:

[EAGAIN] or [EWOULDBLOCK]	The file descriptor is for a socket, is marked O_NONBLOCK, and write would block.
[ECONNRESET]	A write was attempted on a socket that is not connected.
[EPIPE]	A write was attempted on a socket that is shut down for writing, or is no longer connected. In the latter case, if the socket is of type SOCK_STREAM, the SIGPIPE signal is generated to the calling process.

XSI The *write*( ) and *pwrite*( ) functions may fail if:

XSR [EINVAL]	The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.
[EIO]	A physical I/O error has occurred.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[ENXIO]	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
XSR [ENXIO]	A hangup occurred on the STREAM being written to.

XSR A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, *errno* is set to the value included in the error message.

The *write*( ) function may fail if:

[EACCES]	A write was attempted on a socket and the calling process does not have appropriate privileges.
[ENETDOWN]	A write was attempted on a socket and the local network interface used to reach the destination is down.
[ENETUNREACH]	A write was attempted on a socket and no route to the network is present.

XSI The *pwrite*( ) function shall fail and the file pointer remain unchanged if:

XSI [EINVAL]	The *offset* argument is invalid. The value is negative.
XSI [ESPIPE]	*fildes* is associated with a pipe or FIFO.

EXAMPLES

Writing from a Buffer

The following example writes data from the buffer pointed to by *buf* to the file associated with the file descriptor *fd*.

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);

bytes_written = write(fd, buf, nbytes);
...
```

APPLICATION USAGE

None.

RATIONALE

See also the RATIONALE section in *read*( ).

An attempt to write to a pipe or FIFO has several major characteristics:

- *Atomic/non-atomic*: A write is atomic if the whole amount written in one operation is not interleaved with data from any other process. This is useful when there are multiple writers sending data to a single reader. Applications need to know how large a write request can be expected to be performed atomically. This maximum is called {PIPE_BUF}. This volume of IEEE Std 1003.1-200x does not say whether write requests for more than {PIPE_BUF} bytes are atomic, but requires that writes of {PIPE_BUF} or fewer bytes shall be atomic.

- *Blocking/immediate*: Blocking is only possible with O_NONBLOCK clear. If there is enough space for all the data requested to be written immediately, the implementation should do so. Otherwise, the process may block; that is, pause until enough space is available for writing. The effective size of a pipe or FIFO (the maximum amount that can be written in one operation without blocking) may vary dynamically, depending on the implementation, so it is not possible to specify a fixed value for it.

- *Complete/partial/deferred*: A write request:

  ```
  int fildes;
  size_t nbyte;
  ssize_t ret;
  char *buf;

  ret = write(fildes, buf, nbyte);
  ```

  may return:

complete	$\mathit{ret}=\mathit{nbyte}$
partial	$\mathit{ret}<\mathit{nbyte}$ This shall never happen if $\mathit{nbyte}\leq$ {PIPE_BUF}. If it does happen (with $\mathit{nbyte}>$ {PIPE_BUF}), this volume of IEEE Std 1003.1-200x does not guarantee

atomicity, even if *ret*≤{PIPE_BUF}, because atomicity is guaranteed according to the amount *requested*, not the amount *written*.

deferred: *ret*=−1, *errno*=[EAGAIN]

This error indicates that a later request may succeed. It does not indicate that it *shall* succeed, even if *nbyte*≤{PIPE_BUF}, because if no process reads from the pipe or FIFO, the write never succeeds. An application could usefully count the number of times [EAGAIN] is caused by a particular value of *nbyte*>{PIPE_BUF} and perhaps do later writes with a smaller value, on the assumption that the effective size of the pipe may have decreased.

Partial and deferred writes are only possible with O_NONBLOCK set.

The relations of these properties are shown in the following tables:

Write to a Pipe or FIFO with O_NONBLOCK *clear*

Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	Atomic blocking *nbyte*	Atomic blocking *nbyte*	Atomic immediate *nbyte*
nbyte>{PIPE_BUF}	Blocking *nbyte*	Blocking *nbyte*	Blocking *nbyte*

If the O_NONBLOCK flag is clear, a write request shall block if the amount writable immediately is less than that requested. If the flag is set (by *fcntl*( )), a write request shall never block.

Write to a Pipe or FIFO with O_NONBLOCK *set*

Immediately Writable:	**None**	**Some**	*nbyte*
nbyte≤{PIPE_BUF}	−1, [EAGAIN]	−1, [EAGAIN]	Atomic *nbyte*
nbyte>{PIPE_BUF}	−1, [EAGAIN]	<*nbyte* or −1, [EAGAIN]	≤*nbyte* or −1, [EAGAIN]

There is no exception regarding partial writes when O_NONBLOCK is set. With the exception of writing to an empty pipe, this volume of IEEE Std 1003.1-200x does not specify exactly when a partial write is performed since that would require specifying internal details of the implementation. Every application should be prepared to handle partial writes when O_NONBLOCK is set and the requested amount is greater than {PIPE_BUF}, just as every application should be prepared to handle partial writes on other kinds of file descriptors.

The intent of forcing writing at least one byte if any can be written is to assure that each write makes progress if there is any room in the pipe. If the pipe is empty, {PIPE_BUF} bytes must be written; if not, at least some progress must have been made.

Where this volume of IEEE Std 1003.1-200x requires −1 to be returned and *errno* set to [EAGAIN], most historical implementations return zero (with the O_NDELAY flag set, which is the historical predecessor of O_NONBLOCK, but is not itself in this volume of IEEE Std 1003.1-200x). The error indications in this volume of IEEE Std 1003.1-200x were chosen so that an application can distinguish these cases from end-of-file. While *write*( ) cannot receive an indication of end-of-file, *read*( ) can, and the two functions have similar return values. Also, some existing systems (for example, Eighth Edition) permit a write of zero bytes to mean that the reader should get an end-of-file indication; for those systems, a return value of zero from *write*( ) indicates a successful write of an end-of-file indication.

Implementations are allowed, but not required, to perform error checking for *write*( ) requests of zero bytes.

The concept of a {PIPE_MAX} limit (indicating the maximum number of bytes that can be written to a pipe in a single operation) was considered, but rejected, because this concept would unnecessarily limit application writing.

See also the discussion of O_NONBLOCK in *read*( ).

Writes can be serialized with respect to other reads and writes. If a *read*( ) of file data can be proven (by any means) to occur after a *write*( ) of the data, it must reflect that *write*( ), even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from *write*( ) calls to subsequent *read*( ) calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics.

Note that this is specified in terms of *read*( ) and *write*( ). The XSI extensions *readv*( ) and *writev*( ) also obey these semantics. A new ''high-performance'' write analog that did not follow these serialization requirements would also be permitted by this wording. This volume of IEEE Std 1003.1-200x is also silent about any effects of application-level caching (such as that done by *stdio*).

This volume of IEEE Std 1003.1-200x does not specify the value of the file offset after an error is returned; there are too many cases. For programming errors, such as [EBADF], the concept is meaningless since no file is involved. For errors that are detected immediately, such as [EAGAIN], clearly the pointer should not change. After an interrupt or hardware error, however, an updated value would be very useful and is the behavior of many implementations.

This volume of IEEE Std 1003.1-200x does not specify behavior of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *creat*( ), *dup*( ), *fcntl*( ), *getrlimit*( ), *lseek*( ), *open*( ), *pipe*( ), *ulimit*( ), *writev*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<stropts.h>**, **<sys/uio.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

The *pwrite*( ) function is added.

Issue 6

The DESCRIPTION states that the *write*( ) function does not block the thread. Previously this said ''process'' rather than ''thread''.

The DESCRIPTION and ERRORS sections are updated so that references to STREAMS are marked as part of the XSI STREAMS Option Group.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The DESCRIPTION now states that if *write*( ) is interrupted by a signal after it has successfully written some data, it returns the number of bytes written. In earlier versions of this volume of IEEE Std 1003.1-200x, it was optional whether *write*( ) returned the number of bytes written, or whether it returned –1 with *errno* set to [EINTR]. This is a FIPS requirement.
- The following changes are made to support large files:
  - — For regular files, no data transfer occurs past the offset maximum established in the open file description associated with the *fildes*.
  - — A second [EFBIG] error condition is added.
- The [EIO] error condition is added.
- The [EPIPE] error condition is added for when a pipe has only one end open.
- The [ENXIO] optional error condition is added.

Text referring to sockets is added to the DESCRIPTION.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The effect of reading zero bytes is clarified.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by specifying that *write*( ) results are unspecified for typed memory objects.

The following error conditions are added for operations on sockets: [EAGAIN], [EWOULDBLOCK], [ECONNRESET], [ENOTCONN], and [EPIPE].

The [EIO] error is changed to ''may fail''.

The [ENOBUFS] error is added for sockets.

The following error conditions are added for operations on sockets: [EACCES], [ENETDOWN], and [ENETUNREACH].

The *writev*( ) function is split out into a separate reference page.

NAME

writev — write a vector

SYNOPSIS

XSI

```
#include <sys/uio.h>

ssize_t writev(int fildes, const struct iovec *iov, int iovcnt);
```

DESCRIPTION

The *writev*( ) function shall be equivalent to *write*( ), except as described below. The *writev*( ) function shall gather output data from the *iovcnt* buffers specified by the members of the *iov* array: *iov*[0], *iov*[1], . . ., *iov*[*iovcnt*–1]. The *iovcnt* argument is valid if greater than 0 and less than or equal to {IOV_MAX}, defined in **<limits.h>**.

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. The *writev*( ) function shall always write a complete area before proceeding to the next.

If *fildes* refers to a regular file and all of the *iov_len* members in the array pointed to by *iov* are 0, *writev*( ) shall return 0 and have no other effect. For other file types, the behavior is unspecified.

If the sum of the *iov_len* values is greater than {SSIZE_MAX}, the operation shall fail and no data shall be transferred.

RETURN VALUE

Upon successful completion, *writev*( ) shall return the number of bytes actually written. Otherwise, it shall return a value of –1, the file-pointer shall remain unchanged, and *errno* shall be set to indicate an error.

ERRORS

Refer to *write*( ).

In addition, the *writev*( ) function shall fail if:

[EINVAL] The sum of the *iov_len* values in the *iov* array would overflow an **ssize_t**.

The *writev*( ) function may fail and set *errno* to:

[EINVAL] The *iovcnt* argument was less than or equal to 0, or greater than {IOV_MAX}.

EXAMPLES

Writing Data from an Array

The following example writes data from the buffers specified by members of the *iov* array to the file associated with the file descriptor *fd*.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
...
ssize_t bytes_written;
int fd;
char *buf0 = "short string\n";
char *buf1 = "This is a longer string\n";
char *buf2 = "This is the longest string in this example\n";
int iovcnt;
struct iovec iov[3];
```

```
        iov[0].iov_base = buf0;
        iov[0].iov_len = strlen(buf0);
        iov[1].iov_base = buf1;
        iov[1].iov_len = strlen(buf1);
        iov[2].iov_base = buf2;
        iov[2].iov_len = strlen(buf2);
        ...
        iovcnt = sizeof(iov) / sizeof(struct iovec);

        bytes_written = writev(fd, iov, iovcnt);
        ...
```

APPLICATION USAGE

None.

RATIONALE

Refer to *write*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

readv( ), *write*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<limits.h>**, **<sys/uio.h>**

CHANGE HISTORY

First released in Issue 4, Version 2.

Issue 6

Split out from the *write*( ) reference page.

NAME

wscanf — convert formatted wide-character input

SYNOPSIS

```
#include <stdio.h>
#include <wchar.h>

int wscanf(const wchar_t *restrict format, ... );
```

DESCRIPTION

Refer to *fwscanf*( ).

NAME

y0, y1, yn — Bessel functions of the second kind

SYNOPSIS

XSI

```
#include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);
```

DESCRIPTION

The *y0*( ), *y1*( ), and *yn*( ) functions shall compute Bessel functions of *x* of the second kind of orders 0, 1, and *n*, respectively.

An application wishing to check for error situations should set *errno* to zero and call *feclearexcept*(FE_ALL_EXCEPT) before calling these functions. On return, if *errno* is non-zero or *fetestexcept*(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW) is non-zero, an error has occurred.

RETURN VALUE

Upon successful completion, these functions shall return the relevant Bessel value of *x* of the second kind.

If *x* is NaN, NaN shall be returned.

If the *x* argument to these functions is negative, −HUGE_VAL or NaN shall be returned, and a domain error may occur.

If *x* is 0.0, −HUGE_VAL shall be returned and a range error may occur.

If the correct result would cause underflow, 0.0 shall be returned and a range error may occur.

If the correct result would cause overflow, −HUGE_VAL or 0.0 shall be returned and a range error may occur.

ERRORS

These functions may fail if:

Domain Error — The value of *x* is negative.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [EDOM]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the invalid floating-point exception shall be raised.

Range Error — The value of *x* is 0.0, or the correct result would cause overflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the overflow floating-point exception shall be raised.

Range Error — The value of *x* is too large in magnitude, or the correct result would cause underflow.

If the integer expression (math_errhandling & MATH_ERRNO) is non-zero, then *errno* shall be set to [ERANGE]. If the integer expression (math_errhandling & MATH_ERREXCEPT) is non-zero, then the underflow floating-point exception shall be raised.

EXAMPLES

None.

APPLICATION USAGE

On error, the expressions (math_errhandling & MATH_ERRNO) and (math_errhandling & MATH_ERREXCEPT) are independent of each other, but at least one of them must be non-zero.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

feclearexcept( ), *fetestexcept*( ), *isnan*( ), *j0*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.18, Treatment of Error Conditions for Mathematical Functions, **<math.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated to indicate how an application should check for an error. This text was previously published in the APPLICATION USAGE section.

Issue 6

The DESCRIPTION is updated to avoid use of the term ‘‘must’’ for application requirements.

The RETURN VALUE and ERRORS sections are reworked for alignment of the error handling with the ISO/IEC 9899:1999 standard.

Index

Chapter 1

Introduction

1.1 Scope

The scope of IEEE Std 1003.1-200x is described in the Base Definitions volume of IEEE Std 1003.1-200x.

1.2 Conformance

Conformance requirements for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance.

1.3 Normative References

Normative references for IEEE Std 1003.1-200x are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.4 Change History

Change history is described in the Rationale (Informative) volume of IEEE Std 1003.1-200x, and in the CHANGE HISTORY section of reference pages.

1.5 Terminology

This section appears in the Base Definitions volume of IEEE Std 1003.1-200x, but is repeated here for convenience:

For the purposes of IEEE Std 1003.1-200x, the following terminology definitions apply:

can

Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to IEEE Std 1003.1-200x. An application can rely on the existence of the feature or behavior.

implementation-defined

Describes a value or behavior that is not defined by IEEE Std 1003.1-200x but is selected by an implementor. The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations.

The implementor shall document such a value or behavior so that it can be used correctly by an application.

legacy

Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

may

Describes a feature or behavior that is optional for an implementation that conforms to IEEE Std 1003.1-200x. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

To avoid ambiguity, the opposite of *may* is expressed as *need not*, instead of *may not*.

shall

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is mandatory. An application can rely on the existence of the feature or behavior.

For an application or user, describes a behavior that is mandatory.

should

For an implementation that conforms to IEEE Std 1003.1-200x, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations.

For an application, describes a feature or behavior that is recommended programming practice for optimum portability.

undefined

Describes the nature of a value or behavior not defined by IEEE Std 1003.1-200x which results from use of an invalid program construct or invalid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by IEEE Std 1003.1-200x which results from use of a valid program construct or valid data input.

The value or behavior may vary among implementations that conform to IEEE Std 1003.1-200x. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

1.6 Definitions

Concepts and definitions are defined in the Base Definitions volume of IEEE Std 1003.1-200x.

1.7 Relationship to Other Documents

1.7.1 System Interfaces

This subsection describes some of the features provided by the System Interfaces volume of IEEE Std 1003.1-200x that are assumed to be globally available by all systems conforming to this volume of IEEE Std 1003.1-200x. This subsection does not attempt to detail all of the features defined in the System Interfaces volume of IEEE Std 1003.1-200x that are required by all of the utilities defined in this volume of IEEE Std 1003.1-200x; the utility and function descriptions point out additional functionality required to provide the corresponding specific features needed by each.

The following subsections describe frequently used concepts. Many of these concepts are described in the Base Definitions volume of IEEE Std 1003.1-200x. Utility and function description statements override these defaults when appropriate.

1.7.1.1 Process Attributes

The following process attributes, as described in the System Interfaces volume of IEEE Std 1003.1-200x, are assumed to be supported for all processes in this volume of IEEE Std 1003.1-200x:

Controlling Terminal
Current Working Directory
Effective Group ID
Effective User ID
File Descriptors
File Mode Creation Mask
Process Group ID
Process ID
Real Group ID
Real User ID
Root Directory
Saved Set-Group-ID
Saved Set-User-ID
Session Membership
Supplementary Group IDs

A conforming implementation may include additional process attributes.

1.7.1.2 Concurrent Execution of Processes

The following functionality of the *fork*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x shall be available on all systems conforming to this volume of IEEE Std 1003.1-200x:

1. Independent processes shall be capable of executing independently without either process terminating.

2. A process shall be able to create a new process with all of the attributes referenced in Section 1.7.1.1, determined according to the semantics of a call to the *fork*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x followed by a call in the child process to one of the *exec* functions defined in the System Interfaces volume of IEEE Std 1003.1-200x.

1.7.1.3 *File Access Permissions*

The file access control mechanism described by the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.4, File Access Permissions shall apply to all files on an implementation conforming to this volume of IEEE Std 1003.1-200x.

1.7.1.4 *File Read, Write, and Creation*

If a file that does not exist is to be written, it shall be created as described below, unless the utility description states otherwise.

When a file that does not exist is created, the following features defined in the System Interfaces volume of IEEE Std 1003.1-200x shall apply unless the utility or function description states otherwise:

1. The user ID of the file shall be set to the effective user ID of the calling process.
2. The group ID of the file shall be set to the effective group ID of the calling process or the group ID of the directory in which the file is being created.
3. If the file is a regular file, the permission bits of the file shall be set to:

   S_IROTH | S_IWOTH | S_IRGRP | S_IWGRP | S_IRUSR | S_IWUSR

   (see the description of *File Modes* in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers, **<sys/stat.h>**) except that the bits specified by the file mode creation mask of the process shall be cleared. If the file is a directory, the permission bits shall be set to:

   S_IRWXU | S_IRWXG | S_IRWXO

   except that the bits specified by the file mode creation mask of the process shall be cleared.
4. The *st_atime*, *st_ctime*, and *st_mtime* fields of the file shall be updated as specified in the System Interfaces volume of IEEE Std 1003.1-200x, Section 2.5, Standard I/O Streams.
5. If the file is a directory, it shall be an empty directory; otherwise, the file shall have length zero.
6. If the file is a symbolic link, the effect shall be undefined unless the {POSIX2_SYMLINKS} variable is in effect for the directory in which the symbolic link would be created.
7. Unless otherwise specified, the file created shall be a regular file.

When an attempt is made to create a file that already exists, the action shall depend on the type of the file the utility is trying to create and on the type of the existing file as shown in Table 1-1 (on page 2205).

Table 1-1 Actions when Creating a File that Already Exists

		New Type												Function
	Existing Type	**A**	**B**	**C**	**D**	**F**	**L**	**M**	**P**	**Q**	**R**	**S**	**T**	**Creating New**
A	*fattach*( )-ed STREAM	—	—	—	—	—	—	—	—	—	OF	—	—	*fattach*( )
B	Block Special	—	—	—	—	—	—	—	—	—	OF	—	—	*mknod*( )**
C	Character Special	—	—	—	—	—	—	—	—	—	OF	—	—	*mknod*( )**
D	Directory	—	—	—	F	F	—	—	—	—	OD	—	—	*mkdir*( )
F	FIFO Special File	—	—	—	F	F	—	—	—	—	O	—	—	*mkfifo*( )
L	Symbolic Link	FL	FL	FL	FL	FL	FL	FL	FL	FL	FL	FL	FL	*symlink*( )
M	Shared Memory	—	—	—	—	—	—	—	—	—	—	—	—	*shm_open*( )
P	Semaphore	—	—	—	—	—	—	—	—	—	—	—	—	*sem_open*( )
Q	Message Queue	—	—	—	—	—	—	—	—	—	—	—	—	*mq_open*( )
R	Regular File	—	—	—	F	F	—	—	—	—	RF	—	—	*open*( )
S	Socket	—	—	—	—	—	—	—	—	—	—	—	—	*bind*( )
T	Typed Memory	—	—	—	—	—	—	—	—	—	—	—	—	*
	None.	—	—	—	NF	NF	NF	—	—	—	CF	NF	—	

The following codes are used in Table 1-1:

CF Create a new file as defined in Section 1.7.1.4 (on page 2204), items 1 through 7.

F Fail. When attempting to create a directory or FIFO special file, and the existing file is a directory, FIFO special file, or regular file, the attempt shall fail and the utility shall either continue with its operation or exit immediately with a non-zero exit status, depending on the description of the utility.

FL Follow link. Unless otherwise specified, the symbolic links shall be followed as specified for pathname resolution, and the operation performed shall be as if the target of the symbolic link (after all resolution) had been named. If the target of the symbolic link does not exist, it shall be as if that nonexistent target had been named directly.

NF Create a new file as described by the appropriate function.

O Open FIFO. When attempting to create a regular file, and the existing file is a FIFO special file:

1. If the FIFO is not already open for reading, the attempt shall block until the FIFO is opened for reading.
2. Once the FIFO is open for reading, the utility shall open the FIFO for writing and continue with its operation.

OD The directory shall be opened.

OF The named file shall be opened with the consequences defined for that file type.

RF Regular file. When attempting to create a regular file, and the existing file is a regular file:

1. The user ID, group ID, and permission bits of the file shall not be changed.
2. The file shall be truncated to zero length.
3. The *st_ctime* and *st_mtime* fields shall be marked for update.

— The effect is implementation-defined unless specified by the utility description.

* There is no portable way to create a file of this type.

** Not portable.

When a file is to be appended, the file shall be opened in a manner equivalent to using the O_APPEND flag, without the O_TRUNC flag, in the *open*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x.

When a file is to be read or written, the file shall be opened with an access mode corresponding to the operation to be performed. If file access permissions deny access, the requested operation shall fail.

1.7.1.5 File Removal

When a directory that is the root directory or current working directory of any process is removed, the effect is implementation-defined. If file access permissions deny access, the requested operation shall fail. Otherwise, when a file is removed:

1. Its directory entry shall be removed from the file system.
2. The link count of the file shall be decremented.
3. If the file is an empty directory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.143, Empty Directory):
    a. If no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible.
    b. If one or more processes have the directory open, the directory contents shall be preserved until all references to the file have been closed.
4. If the file is a directory that is not empty, the *st_ctime* field shall be marked for update.
5. If the file is not a directory:
    a. If the link count becomes zero:
        i. If no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible.
        ii. If one or more processes have the file open, the file contents shall be preserved until all references to the file have been closed.
    b. If the link count is not reduced to zero, the *st_ctime* field shall be marked for update.
6. The *st_ctime* and *st_mtime* fields of the containing directory shall be marked for update.

1.7.1.6 File Time Values

All files shall have the three time values described by the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.7, File Times Update.

1.7.1.7 File Contents

When a reference is made to the contents of a file, *pathname*, this means the equivalent of all of the data placed in the space pointed to by *buf* when performing the *read*( ) function calls in the following operations defined in the System Interfaces volume of IEEE Std 1003.1-200x:

```
while (read (fildes, buf, nbytes) > 0)
    ;
```

If the file is indicated by a pathname *pathname*, the file descriptor shall be determined by the equivalent of the following operation defined in the System Interfaces volume of IEEE Std 1003.1-200x:

```
fildes = open (pathname, O_RDONLY);
```

The value of *nbytes* in the above sequence is unspecified; if the file is of a type where the data returned by *read*( ) would vary with different values, the value shall be one that results in the most data being returned.

If the *read*( ) function calls would return an error, it is unspecified whether the contents of the file are considered to include any data from offsets in the file beyond where the error would be returned.

1.7.1.8 Pathname Resolution

The pathname resolution algorithm, described by the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.11, Pathname Resolution, shall be used by implementations conforming to this volume of IEEE Std 1003.1-200x; see also the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.5, File Hierarchy.

1.7.1.9 Changing the Current Working Directory

When the current working directory (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.436, Working Directory) is to be changed, unless the utility or function description states otherwise, the operation shall succeed unless a call to the *chdir*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x would fail when invoked with the new working directory pathname as its argument.

1.7.1.10 Establish the Locale

The functionality of the *setlocale*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x shall be available on all systems conforming to this volume of IEEE Std 1003.1-200x; that is, utilities that require the capability of establishing an international operating environment shall be permitted to set the specified category of the international environment.

1.7.1.11 Actions Equivalent to Functions

Some utility descriptions specify that a utility performs actions equivalent to a function defined in the System Interfaces volume of IEEE Std 1003.1-200x. Such specifications require only that the external effects be equivalent, not that any effect within the utility and visible only to the utility be equivalent.

1.7.2 Concepts Derived from the ISO C Standard

Some of the standard utilities perform complex data manipulation using their own procedure and arithmetic languages, as defined in their EXTENDED DESCRIPTION or OPERANDS sections. Unless otherwise noted, the arithmetic and semantic concepts (precision, type conversion, control flow, and so on) shall be equivalent to those defined in the ISO C standard, as described in the following sections. Note that there is no requirement that the standard utilities be implemented in any particular programming language.

1.7.2.1 Arithmetic Precision and Operations

Integer variables and constants, including the values of operands and option-arguments, used by the standard utilities listed in this volume of IEEE Std 1003.1-200x shall be implemented as equivalent to the ISO C standard **signed long** data type; floating point shall be implemented as equivalent to the ISO C standard **double** type. Conversions between types shall be as described in the ISO C standard. All variables shall be initialized to zero if they are not otherwise assigned by the input to the application.

Arithmetic operators and functions shall be implemented as equivalent to those in the cited ISO C standard section, as listed in Table 1-2 (on page 2209).

Table 1-2 ISO C Standard Operators and Functions

Operation	ISO C Standard Equivalent Reference
()	Section 6.5.1, Primary Expressions
postfix ++ postfix --	Section 6.5.2, Postfix Operators
unary + unary - prefix ++ prefix -- ~ ! *sizeof*()	Section 6.5.3, Unary Operators
* / %	Section 6.5.5, Multiplicative Operators
+ -	Section 6.5.6, Additive Operators
<< >>	Section 6.5.7, Bitwise Shift Operators
<, <= >, >=	Section 6.5.8, Relational Operators
== !=	Section 6.5.9, Equality Operators
&	Section 6.5.10, Bitwise AND Operator
^	Section 6.5.11, Bitwise Exclusive OR Operator
|	Section 6.5.12, Bitwise Inclusive OR Operator
&&	Section 6.5.13, Logical AND Operator
||	Section 6.5.14, Logical OR Operator
expr?*expr*:*expr*	Section 6.5.15, Conditional Operator
=, *=, /=, %=, +=, -= <<=, >>=, &=, ^=, |=	Section 6.5.16, Assignment Operators
if () **if** () . . . **else** **switch** ()	Section 6.8.4, Selection Statements
while () **do** . . . **while** () **for** ()	Section 6.8.5, Iteration Statements
goto **continue** **break** **return**	Section 6.8.6, Jump Statements

The evaluation of arithmetic expressions shall be equivalent to that described in Section 6.5, Expressions, of the ISO C standard.

1.7.2.2 Mathematical Functions

Any mathematical functions with the same names as those in the following sections of the ISO C standard:

- Section 7.12, Mathematics, `<math.h>`
- Section 7.20.2, Pseudo-Random Sequence Generation Functions

shall be implemented to return the results equivalent to those returned from a call to the corresponding function described in the ISO C standard.

1.8 Portability

Some of the utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x and functions in the System Interfaces volume of IEEE Std 1003.1-200x describe functionality that might not be fully portable to systems meeting the requirements for POSIX conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance).

Where optional, enhanced, or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the option, extension, or warning (see Section 1.8.1). For maximum portability, an application should avoid such functionality.

Unless the primary task of a utility is to produce textual material on its standard output, application developers should not rely on the format or content of any such material that may be produced. Where the primary task *is* to provide such material, but the output format is incompletely specified, the description is marked with the OF margin code and shading. Application developers are warned not to expect that the output of such an interface on one system is any guide to its behavior on another system.

1.8.1 Codes

Codes and their meanings are listed in the Base Definitions volume of IEEE Std 1003.1-200x, but are repeated here for convenience:

ADV Advisory Information
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ADV margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ADV margin legend.

AIO Asynchronous Input and Output
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the AIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the AIO margin legend.

BAR Barriers
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the BAR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the BAR margin legend.

BE Batch Environment Services and Utilities

The functionality described is optional.

Where applicable, utilities are marked with the BE margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the BE margin legend.

CD C-Language Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the CD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the CD margin legend.

CPT Process CPU-Time Clocks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CPT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CPT margin legend.

CS Clock Selection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the CS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the CS margin legend.

CX Extension to the ISO C standard

The functionality described is an extension to the ISO C standard. Application writers may make use of an extension as it is supported on all IEEE Std 1003.1-200x-conforming systems.

With each function or header from the ISO C standard, a statement to the effect that ‘‘any conflict is unintentional’’ is included. That is intended to refer to a direct conflict. IEEE Std 1003.1-200x acts in part as a profile of the ISO C standard, and it may choose to further constrain behaviors allowed to vary by the ISO C standard. Such limitations are not considered conflicts.

FD FORTRAN Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the FD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FD margin legend.

FR FORTRAN Runtime Utilities

The functionality described is optional.

Where applicable, utilities are marked with the FR margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the FR margin legend.

FSC File Synchronization

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the FSC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the FSC margin legend.

IP6 IPV6
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the IP6 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the IP6 margin legend.

MC1 Advisory Information and either Memory Mapped Files or Shared Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC1 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC1 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code Notation.

MC2 Memory Mapped Files, Shared Memory Objects, or Memory Protection
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

This is a shorthand notation for combinations of multiple option codes.

Where applicable, functions are marked with the MC2 margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MC2 margin legend.

Refer to the Base Definitions volume of IEEE Std 1003.1-200x, Section 1.5.2, Margin Code Notation.

MF Memory Mapped Files
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MF margin legend.

ML Process Memory Locking
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the ML margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the ML margin legend.

MLR Range Memory Locking
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MLR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MLR margin legend.

MON Monotonic Clock

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MON margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MON margin legend.

MPR Memory Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MPR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MPR margin legend.

MSG Message Passing

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MSG margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MSG margin legend.

MX IEC 60559 Floating-Point Option

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the MX margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the MX margin legend.

OB Obsolescent

The functionality described may be withdrawn in a future version of this volume of IEEE Std 1003.1-200x. Strictly Conforming POSIX Applications and Strictly Conforming XSI Applications shall not use obsolescent features.

OF Output Format Incompletely Specified

The functionality described is an XSI extension. The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OH Optional Header

In the SYNOPSIS section of some interfaces in the System Interfaces volume of IEEE Std 1003.1-200x an included header is marked as in the following example:

OH
```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

This indicates that the marked header is not required on XSI-conformant systems.

PIO Prioritized Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PIO margin legend.

PS Process Scheduling

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the PS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the PS margin legend.

RS Raw Sockets

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RS margin legend.

RTS Realtime Signals Extension

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the RTS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the RTS margin legend.

SD Software Development Utilities

The functionality described is optional.

Where applicable, utilities are marked with the SD margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the SD margin legend.

SEM Semaphores

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SEM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SEM margin legend.

SHM Shared Memory Objects

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SHM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SHM margin legend.

SIO Synchronized Input and Output

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SIO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SIO margin legend.

SPI Spin Locks

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPI margin legend.

SPN Spawn
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SPN margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SPN margin legend.

SS Process Sporadic Server
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the SS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the SS margin legend.

TCT Thread CPU-Time Clocks
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TCT margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TCT margin legend.

TEF Trace Event Filter
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TEF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TEF margin legend.

THR Threads
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the THR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the THR margin legend.

TMO Timeouts
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMO margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMO margin legend.

TMR Timers
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TMR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TMR margin legend.

TPI Thread Priority Inheritance

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPI margin legend.

TPP Thread Priority Protection

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPP margin legend.

TPS Thread Execution Scheduling

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TPS margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TPS margin legend.

TRC Trace

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRC margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRC margin legend.

TRI Trace Inherit

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRI margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRI margin legend.

TRL Trace Log

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TRL margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TRL margin legend.

TSA Thread Stack Address Attribute

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSA margin legend for the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSA margin legend.

TSF Thread-Safe Functions

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSF margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSF margin legend.

TSH Thread Process-Shared Synchronization
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSH margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSH margin legend.

TSP Thread Sporadic Server
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSP margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSP margin legend.

TSS Thread Stack Address Size
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TSS margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TSS margin legend.

TYM Typed Memory Objects
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the TYM margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the TYM margin legend. |

UP User Portability Utilities
The functionality described is optional.

Where applicable, utilities are marked with the UP margin legend in the SYNOPSIS section. Where additional semantics apply to a utility, the material is identified by use of the UP margin legend.

XSI Extension
The functionality described is an XSI extension. Functionality marked XSI is also an extension to the ISO C standard. Application writers may confidently make use of an extension on all systems supporting the X/Open System Interfaces Extension.

If an entire SYNOPSIS section is shaded and marked XSI, all the functionality described in that reference page is an extension. See the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.439, XSI.

XSR XSI STREAMS
The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the XSR margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the XSR margin legend.

1.9 Utility Limits

This section lists magnitude limitations imposed by a specific implementation. The braces notation, {LIMIT}, is used in this volume of IEEE Std 1003.1-200x to indicate these values, but the braces are not part of the name.

Table 1-3 Utility Limit Minimum Values

Name	Description	Value
{POSIX2_BC_BASE_MAX}	The maximum *obase* value allowed by the *bc* utility.	99
{POSIX2_BC_DIM_MAX}	The maximum number of elements permitted in an array by the *bc* utility.	2048
{POSIX2_BC_SCALE_MAX}	The maximum *scale* value allowed by the *bc* utility.	99
{POSIX2_BC_STRING_MAX}	The maximum length of a string constant accepted by the *bc* utility.	1000
{POSIX2_COLL_WEIGHTS_MAX}	The maximum number of weights that can be assigned to an entry of the *LC_COLLATE* **order** keyword in the locale definition file; see the **border_start** keyword in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, LC_COLLATE.	2
{POSIX2_EXPR_NEST_MAX}	The maximum number of expressions that can be nested within parentheses by the *expr* utility.	32
{POSIX2_LINE_MAX}	Unless otherwise noted, the maximum length, in bytes, of the input line of a utility (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing newline.	2048
{POSIX2_RE_DUP_MAX}	The maximum number of repeated occurrences of a BRE permitted when using the interval notation \\{*m*,*n*\\}; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.6, BREs Matching Multiple Characters.	255
{POSIX2_VERSION}	This value indicates the version of the utilities in this volume of IEEE Std 1003.1-200x that are provided by the implementation. It changes with each published version.	200xxxL

The values specified in Table 1-3 represent the lowest values conforming implementations shall provide and, consequently, the largest values on which an application can rely without further enquiries, as described below. These values shall be accessible to applications via the *getconf* utility (see *getconf* (on page 2683)) and through the *sysconf*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. The literal names shown in Table 1-3 apply only to the *getconf* utility; the high-level language binding describes the exact form of each name to be used by the interfaces in that binding.

Implementations may provide more liberal, or less restrictive, values than shown in Table 1-3. These possibly more liberal values are accessible using the symbols in Table 1-4 (on page 2219).

The *sysconf*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x or the *getconf* utility return the value of each symbol on each specific implementation. The value so retrieved is the largest, or most liberal, value that is available throughout the session lifetime, as determined at session creation. The literal names shown in the table apply only to the *getconf* utility; the high-level language binding describes the exact form of each name to be used by the interfaces in that binding.

All numeric limits defined by the System Interfaces volume of IEEE Std 1003.1-200x, such as {PATH_MAX}, shall also apply to this volume of IEEE Std 1003.1-200x. All the utilities defined by this volume of IEEE Std 1003.1-200x are implicitly limited by these values, unless otherwise noted in the utility descriptions.

It is not guaranteed that the application can actually reach the specified limit of an implementation in any given case, or at all, as a lack of virtual memory or other resources may prevent this. The limit value indicates only that the implementation does not specifically impose any arbitrary, more restrictive limit.

Table 1-4 Symbolic Utility Limits

Name	Description	Minimum Value
{BC_BASE_MAX}	The maximum *obase* value allowed by the *bc* utility.	{POSIX2_BC_BASE_MAX}
{BC_DIM_MAX}	The maximum number of elements permitted in an array by the *bc* utility.	{POSIX2_BC_DIM_MAX}
{BC_SCALE_MAX}	The maximum *scale* value allowed by the *bc* utility.	{POSIX2_BC_SCALE_MAX}
{BC_STRING_MAX}	The maximum length of a string constant accepted by the *bc* utility.	{POSIX2_BC_STRING_MAX}
{COLL_WEIGHTS_MAX}	The maximum number of weights that can be assigned to an entry of the *LC_COLLATE* **order** keyword in the locale definition file; see the **order_start** keyword in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.2, LC_COLLATE.	{POSIX2_COLL_WEIGHTS_MAX}
{EXPR_NEST_MAX}	The maximum number of expressions that can be nested within parentheses by the *expr* utility.	{POSIX2_EXPR_NEST_MAX}
{LINE_MAX}	Unless otherwise noted, the maximum length, in bytes, of the input line of a utility (either standard input or another file), when the utility is described as	{POSIX2_LINE_MAX}

Name	Description	Minimum Value
	processing text files. The length includes room for the trailing newline.	
{RE_DUP_MAX}	The maximum number of repeated occurrences of a BRE permitted when using the interval notation \{*m*,*n*\}; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.6, BREs Matching Multiple Characters.	{POSIX2_RE_DUP_MAX}

The following value may be a constant within an implementation or may vary from one pathname to another.

{POSIX2_SYMLINKS}
When referring to a directory, the system supports the creation of symbolic links within that directory; for non-directory files, the meaning of {POSIX2_SYMLINKS} is undefined.

1.10 Grammar Conventions

Portions of this volume of IEEE Std 1003.1-200x are expressed in terms of a special grammar notation. It is used to portray the complex syntax of certain program input. The grammar is based on the syntax used by the *yacc* utility. However, it does not represent fully functional *yacc* input, suitable for program use; the lexical processing and all semantic requirements are described only in textual form. The grammar is not based on source used in any traditional implementation and has not been tested with the semantic code that would normally be required to accompany it. Furthermore, there is no implication that the partial *yacc* code presented represents the most efficient, or only, means of supporting the complex syntax within the utility. Implementations may use other programming languages or algorithms, as long as the syntax supported is the same as that represented by the grammar.

The following typographical conventions are used in the grammar; they have no significance except to aid in reading.

- The identifiers for the reserved words of the language are shown with a leading capital letter. (These are terminals in the grammar; for example, **While**, **Case**.)
- The identifiers for terminals in the grammar are all named with uppercase letters and underscores; for example, **NEWLINE**, **ASSIGN_OP**, **NAME**.
- The identifiers for non-terminals are all lowercase.

1.11 Utility Description Defaults

This section describes all of the subsections used within the utility descriptions, including:

- Intended usage of the section
- Global defaults that affect all the standard utilities
- The meanings of notations used in this volume of IEEE Std 1003.1-200x that are specific to individual utility sections

NAME

This section gives the name or names of the utility and briefly states its purpose.

SYNOPSIS

The SYNOPSIS section summarizes the syntax of the calling sequence for the utility, including options, option-arguments, and operands. Standards for utility naming are described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines; for describing the utility's arguments in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.1, Utility Argument Syntax.

DESCRIPTION

The DESCRIPTION section describes the actions of the utility. If the utility has a very complex set of subcommands or its own procedural language, an EXTENDED DESCRIPTION section is also provided. Most explanations of optional functionality are omitted here, as they are usually explained in the OPTIONS section.

As stated in Section 1.7.1.11 (on page 2207), some functions are described in terms of equivalent functionality. When specific functions are cited, the implementation shall provide equivalent functionality including side effects associated with successful execution of the function. The treatment of errors and intermediate results from the individual functions cited is generally not specified by this volume of IEEE Std 1003.1-200x. See the utility's EXIT STATUS and CONSEQUENCES OF ERRORS sections for all actions associated with errors encountered by the utility.

OPTIONS

The OPTIONS section describes the utility options and option-arguments, and how they modify the actions of the utility. Standard utilities that have options either fully comply with the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines or describe all deviations. Apparent disagreements between functionality descriptions in the OPTIONS and DESCRIPTION (or EXTENDED DESCRIPTION) sections are always resolved in favor of the OPTIONS section.

Each OPTIONS section that uses the phrase ''The ... utility shall conform to the Utility Syntax Guidelines ...'' refers only to the use of the utility as specified by this volume of IEEE Std 1003.1-200x; implementation extensions should also conform to the guidelines, but may allow exceptions for historical practice.

Unless otherwise stated in the utility description, when given an option unrecognized by the implementation, or when a required option-argument is not provided, standard utilities shall issue a diagnostic message to standard error and exit with a non-zero exit status.

All utilities in this volume of IEEE Std 1003.1-200x shall be capable of processing arguments using eight-bit transparency.

Default Behavior: When this section is listed as ''None.'', it means that the implementation need not support any options. Standard utilities that do not accept options, but that do accept operands, shall recognize "−−" as a first argument to be discarded.

The requirement for recognizing "−−" is because conforming applications need a way to shield their operands from any arbitrary options that the implementation may provide as an extension. For example, if the standard utility *foo* is listed as taking no options, and the application needed to give it a pathname with a leading hyphen, it could safely do it as:

```
foo −− −myfile
```

and avoid any problems with **−m** used as an extension.

OPERANDS

The OPERANDS section describes the utility operands, and how they affect the actions of the utility. Apparent disagreements between functionality descriptions in the OPERANDS and DESCRIPTION (or EXTENDED DESCRIPTION) sections shall be resolved in favor of the OPERANDS section.

If an operand naming a file can be specified as ′−′, which means to use the standard input instead of a named file, this is explicitly stated in this section. Unless otherwise stated, the use of multiple instances of ′−′ to mean standard input in a single command produces unspecified results.

Unless otherwise stated, the standard utilities that accept operands shall process those operands in the order specified in the command line.

Default Behavior: When this section is listed as ‘‘None.’’, it means that the implementation need not support any operands.

STDIN

The STDIN section describes the standard input of the utility. This section is frequently merely a reference to the following section, as many utilities treat standard input and input files in the same manner. Unless otherwise stated, all restrictions described in the INPUT FILES section shall apply to this section as well.

Use of a terminal for standard input can cause any of the standard utilities that read standard input to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

The specified standard input format of the standard utilities shall not depend on the existence or value of the environment variables defined in this volume of IEEE Std 1003.1-200x, except as provided by this volume of IEEE Std 1003.1-200x.

Default Behavior: When this section is listed as ‘‘Not used.’’, it means that the standard input shall not be read when the utility is used as described by this volume of IEEE Std 1003.1-200x.

INPUT FILES

The INPUT FILES section describes the files, other than the standard input, used as input by the utility. It includes files named as operands and option-arguments as well as other files that are referred to, such as start-up and initialization files, databases, and so on. Commonly-used files are generally described in one place and cross-referenced by other utilities.

All utilities in this volume of IEEE Std 1003.1-200x shall be capable of processing input files using eight-bit transparency.

When a standard utility reads a seekable input file and terminates without an error before it reaches end-of-file, the utility shall ensure that the file offset in the open file description is properly positioned just past the last byte processed by the utility. For files that are not seekable, the state of the file offset in the open file description for that file is unspecified. A conforming application shall not assume that the following three commands are equivalent:

```
tail −n +2 file
(sed −n 1q; cat) < file
cat file | (sed −n 1q; cat)
```

The second command is equivalent to the first only when the file is seekable. The third command leaves the file offset in the open file description in an unspecified state. Other utilities, such as *head*, *read*, and *sh*, have similar properties.

Some of the standard utilities, such as filters, process input files a line or a block at a time and have no restrictions on the maximum input file size. Some utilities may have size limitations that are not as obvious as file space or memory limitations. Such limitations should reflect resource limitations of some sort, not arbitrary limits set by implementors. Implementations shall document those utilities that are limited by constraints other than file system space, available memory, and other limits specifically cited by this volume of IEEE Std 1003.1-200x, and identify what the constraint is and indicate a way of estimating when the constraint would be reached. Similarly, some utilities descend the directory tree (recursively). Implementations shall also document any limits that they may have in descending the directory tree that are beyond limits cited by this volume of IEEE Std 1003.1-200x.

When an input file is described as a *text file*, the utility produces undefined results if given input that is not from a text file, unless otherwise stated. Some utilities (for example, *make*, *read*, *sh*) allow for continued input lines using an escaped <newline> convention; unless otherwise stated, the utility need not be able to accumulate more than {LINE_MAX} bytes from a set of multiple, continued input lines. Thus, for a conforming application the total of all the continued lines in a set cannot exceed {LINE_MAX}. If a utility using the escaped <newline> convention detects an end-of-file condition immediately after an escaped <newline>, the results are unspecified.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation for a description of this notation. The format description is intended to be sufficiently rigorous to allow other applications to generate these input files. However, since <blank>s can legitimately be included in some of the fields described by the standard utilities, particularly in locales other than the POSIX locale, this intent is not always realized.

Default Behavior: When this section is listed as ‘‘None.’’, it means that no input files are required to be supplied when the utility is used as described by this volume of IEEE Std 1003.1-200x.

ENVIRONMENT VARIABLES

The ENVIRONMENT VARIABLES section lists what variables affect the utility’s execution.

The entire manner in which environment variables described in this volume of IEEE Std 1003.1-200x affect the behavior of each utility is described in the ENVIRONMENT VARIABLES section for that utility, in conjunction with the global effects of the *LANG*, *LC_ALL*, and XSI *NLSPATH* environment variables described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables. The existence or value of environment variables described in this volume of

IEEE Std 1003.1-200x shall not otherwise affect the specified behavior of the standard utilities. Any effects of the existence or value of environment variables not described by this volume of IEEE Std 1003.1-200x upon the standard utilities are unspecified.

For those standard utilities that use environment variables as a means for selecting a utility to execute (such as *CC* in *make*), the string provided to the utility is subjected to the path search described for *PATH* in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

All utilities in this volume of IEEE Std 1003.1-200x shall be capable of processing environment variable names and values using eight-bit transparency.

Default Behavior: When this section is listed as ''None.'', it means that the behavior of the utility is not directly affected by environment variables described by this volume of IEEE Std 1003.1-200x when the utility is used as described by this volume of IEEE Std 1003.1-200x.

ASYNCHRONOUS EVENTS

The ASYNCHRONOUS EVENTS section lists how the utility reacts to such events as signals and what signals are caught.

Default Behavior: When this section is listed as ''Default.'', or it refers to ''the standard action for all other signals; see Section 1.11 (on page 2221)'' it means that the action taken as a result of the signal shall be one of the following:

1. The action shall be that inherited from the parent according to the rules of inheritance of signal actions defined in the System Interfaces volume of IEEE Std 1003.1-200x.
2. When no action has been taken to change the default, the default action shall be that specified by the System Interfaces volume of IEEE Std 1003.1-200x.
3. The result of the utility's execution is as if default actions had been taken.

A utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process), and resignal itself.

STDOUT

The STDOUT section completely describes the standard output of the utility. This section is frequently merely a reference to the following section, OUTPUT FILES, because many utilities treat standard output and output files in the same manner.

Use of a terminal for standard output may cause any of the standard utilities that write standard output to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation for a description of this notation.

The specified standard output of the standard utilities shall not depend on the existence or value of the environment variables defined in this volume of IEEE Std 1003.1-200x, except as provided by this volume of IEEE Std 1003.1-200x.

Some of the standard utilities describe their output using the verb *display*, defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.132, Display. Output described in the STDOUT sections of such utilities may be produced using means other than standard output. When standard output is directed to a terminal, the output described shall be written directly to the terminal. Otherwise, the results are undefined.

Default Behavior: When this section is listed as ''Not used.'', it means that the standard output shall not be written when the utility is used as described by this volume of IEEE Std 1003.1-200x.

STDERR

The STDERR section describes the standard error output of the utility. Only those messages that are purposely sent by the utility are described.

Use of a terminal for standard error may cause any of the standard utilities that write standard error output to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

The format of diagnostic messages for most utilities is unspecified, but the language and cultural conventions of diagnostic and informative messages whose format is unspecified by this volume of IEEE Std 1003.1-200x should be affected by the setting of XSI *LC_MESSAGES* and *NLSPATH*.

The specified standard error output of standard utilities shall not depend on the existence or value of the environment variables defined in this volume of IEEE Std 1003.1-200x, except as provided by this volume of IEEE Std 1003.1-200x.

Default Behavior: When this section is listed as ''Used only for diagnostic messages.'', it means that, unless otherwise stated, the diagnostic messages shall be sent to the standard error only when the exit status is non-zero and the utility is used as described by this volume of IEEE Std 1003.1-200x.

When this section is listed as ''Not used.'', it means that the standard error shall not be used when the utility is used as described in this volume of IEEE Std 1003.1-200x.

OUTPUT FILES

The OUTPUT FILES section completely describes the files created or modified by the utility. Temporary or system files that are created for internal usage by this utility or other parts of the implementation (for example, spool, log, and audit files) are not described in this, or any, section. The utilities creating such files and the names of such files are unspecified. If applications are written to use temporary or intermediate files, they should use the *TMPDIR* environment variable, if it is set and represents an accessible directory, to select the location of temporary files.

Implementations shall ensure that temporary files, when used by the standard utilities, are named so that different utilities or multiple instances of the same utility can operate simultaneously without regard to their working directories, or any other process characteristic other than process ID. There are two exceptions to this rule:

1. Resources for temporary files other than the name space (for example, disk space, available directory entries, or number of processes allowed) are not guaranteed.
2. Certain standard utilities generate output files that are intended as input for other utilities (for example, *lex* generates **lex.yy.c**), and these cannot have unique names. These cases are explicitly identified in the descriptions of the respective utilities.

Any temporary file created by the implementation shall be removed by the implementation upon a utility's successful exit, exit because of errors, or before termination by any of the SIGHUP, SIGINT, or SIGTERM signals, unless specified otherwise by the utility description.

Receipt of the SIGQUIT signal should generally cause termination (unless in some debugging mode) that would bypass any attempted recovery actions.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ); see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation for a description of this notation.

Default Behavior: When this section is listed as ‘‘None.’’, it means that no files are created or modified as a consequence of direct action on the part of the utility when the utility is used as described by this volume of IEEE Std 1003.1-200x. However, the utility may create or modify system files, such as log files, that are outside the utility’s normal execution environment.

EXTENDED DESCRIPTION

The EXTENDED DESCRIPTION section provides a place for describing the actions of very complicated utilities, such as text editors or language processors, which typically have elaborate command languages.

Default Behavior: When this section is listed as ‘‘None.’’, no further description is necessary.

EXIT STATUS

The EXIT STATUS section describes the values the utility shall return to the calling program, or shell, and the conditions that cause these values to be returned. Usually, utilities return zero for successful completion and values greater than zero for various error conditions. If specific numeric values are listed in this section, the system shall use those values for the errors described. In some cases, status values are listed more loosely, such as >0. A strictly conforming application shall not rely on any specific value in the range shown and shall be prepared to receive any value in the range.

For example, a utility may list zero as a successful return, 1 as a failure for a specific reason, and >1 as ‘‘an error occurred’’. In this case, unspecified conditions may cause a 2 or 3, or other value, to be returned. A conforming application should be written so that it tests for successful exit status values (zero in this case), rather than relying upon the single specific error value listed in this volume of IEEE Std 1003.1-200x. In that way, it has maximum portability, even on implementations with extensions.

Unspecified error conditions may be represented by specific values not listed in this volume of IEEE Std 1003.1-200x.

CONSEQUENCES OF ERRORS

The CONSEQUENCES OF ERRORS section describes the effects on the environment, file systems, process state, and so on, when error conditions occur. It does not describe error messages produced or exit status values used.

The many reasons for failure of a utility are generally not specified by the utility descriptions. Utilities may terminate prematurely if they encounter: invalid usage of options, arguments, or environment variables; invalid usage of the complex syntaxes expressed in EXTENDED DESCRIPTION sections; difficulties accessing, creating, reading, or writing files; or difficulties associated with the privileges of the process.

The following shall apply to each utility, unless otherwise stated:

- If the requested action cannot be performed on an operand representing a file, directory, user, process, and so on, the utility shall issue a diagnostic message to standard error and continue processing the next operand in sequence, but the final exit status shall be returned as non-zero.

For a utility that recursively traverses a file hierarchy (such as *find* or *chown* −**R**), if the requested action cannot be performed on a file or directory encountered in the hierarchy, the utility shall issue a diagnostic message to standard error and continue processing the remaining files in the hierarchy, but the final exit status shall be returned as non-zero.

- If the requested action characterized by an option or option-argument cannot be performed, the utility shall issue a diagnostic message to standard error and the exit status returned shall be non-zero.
- When an unrecoverable error condition is encountered, the utility shall exit with a non-zero exit status.
- A diagnostic message shall be written to standard error whenever an error condition occurs.

When a utility encounters an error condition several actions are possible, depending on the severity of the error and the state of the utility. Included in the possible actions of various utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; validity checking of the file system or directory.

Default Behavior: When this section is listed as ‘‘Default.’’, it means that any changes to the environment are unspecified.

APPLICATION USAGE

This section is non-normative.

The APPLICATION USAGE section gives advice to the application programmer or user about the way the utility should be used.

EXAMPLES

This section is non-normative.

The EXAMPLES section gives one or more examples of usage, where appropriate. In the event of conflict between an example and a normative part of the specification, the normative material is to be taken as correct.

In all examples, quoting has been used, showing how sample commands (utility names combined with arguments) could be passed correctly to a shell (see *sh*) or as a string to the *system*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. Such quoting would not be used if the utility is invoked using one of the *exec* functions defined in the System Interfaces volume of IEEE Std 1003.1-200x.

RATIONALE

This section is non-normative.

This section contains historical information concerning the contents of this volume of IEEE Std 1003.1-200x and why features were included or discarded by the standard developers.

FUTURE DIRECTIONS

This section is non-normative.

The FUTURE DIRECTIONS section should be used as a guide to current thinking; there is not necessarily a commitment to implement all of these future directions in their entirety.

SEE ALSO

This section is non-normative.

The SEE ALSO section lists related entries.

CHANGE HISTORY

This section is non-normative.

The CHANGE HISTORY section shows the derivation of the description used by this volume of IEEE Std 1003.1-200x and lists the functional differences between Issues 4 and 6.

Certain of the standard utilities describe how they can invoke other utilities or applications, such as by passing a command string to the command interpreter. The external influences (STDIN, ENVIRONMENT VARIABLES, and so on) and external effects (STDOUT, CONSEQUENCES OF ERRORS, and so on) of such invoked utilities are not described in the section concerning the standard utility that invokes them.

1.12 Considerations for Utilities in Support of Files of Arbitrary Size

The following utilities support files of any size up to the maximum that can be created by the implementation. This support includes correct writing of file size-related values (such as file sizes and offsets, line numbers, and block counts) and correct interpretation of command line arguments that contain such values.

basename	Return non-directory portion of pathname.
cat	Concatenate and print files.
cd	Change working directory.
chgrp	Change file group ownership.
chmod	Change file modes.
chown	Change file ownership.
cksum	Write file checksums and sizes.
cmp	Compare two files.
cp	Copy files.
dd	Convert and copy a file.
df	Report free disk space.
dirname	Return directory portion of pathname.
du	Estimate file space usage.
find	Find files.
ln	Link files.
ls	List directory contents.
mkdir	Make directories.
mv	Move files.
pathchk	Check pathnames.
pwd	Return working directory name.

rm	Remove directory entries.
rmdir	Remove directories.
sh	Shell, the standard command language interpreter.
sum	Print checksum and block or byte count of a file.
test	Evaluate expression.
touch	Change file access and modification times.
ulimit	Set or report file size limit.

Exceptions to the requirement that utilities support files of any size up to the maximum are as follows:

1. Uses of files as command scripts, or for configuration or control, are exempt. For example, it is not required that *sh* be able to read an arbitrarily large **.profile**.
2. Shell input and output redirection are exempt. For example, it is not required that the redirections *sum* < *file* or *echo foo* > *file* succeed for an arbitrarily large existing file.

1.13 Built-In Utilities

Any of the standard utilities may be implemented as *regular built-in* utilities within the command language interpreter. This is usually done to increase the performance of frequently used utilities or to achieve functionality that would be more difficult in a separate environment. The utilities named in Table 1-5 are frequently provided in built-in form. All of the utilities named in the table have special properties in terms of command search order within the shell, as described in Section 2.9.1.1 (on page 2249).

Table 1-5 Regular Built-in Utilities

alias	*false*	*jobs*	*true*
bg	*fc*	*kill*	*umask*
cd	*fg*	*newgrp*	*unalias*
command	*getopts*	*read*	*wait*

However, all of the standard utilities, including the regular built-ins in the table, but not the special built-ins described in Section 2.14 (on page 2266), shall be implemented in a manner so that they can be accessed via the *exec* family of functions as defined in the System Interfaces volume of IEEE Std 1003.1-200x and can be invoked directly by those standard utilities that require it (*env*, *find*, *nice*, *nohup*, *time*, *xargs*).

Since *exec*-able versions shall be provided for all utilities except for those listed in Section 2.14 (on page 2266), an application running on a system that conforms to IEEE Std 1003.1-200x can use the *exec* family of functions, in addition to the shell command interface provided by the *system*( ) and *popen*( ) functions, to execute any of these utilities.

Chapter 2

Shell Command Language

This chapter contains the definition of the Shell Command Language.

2.1 Shell Introduction

The shell is a command language interpreter. This chapter describes the syntax of that command language as it is used by the *sh* utility and the *system*( ) and *popen*( ) functions defined in the System Interfaces volume of IEEE Std 1003.1-200x.

The shell operates according to the following general overview of operations. The specific details are included in the cited sections of this chapter.

1. The shell reads its input from a file (see *sh*), from the −**c** option or from the *system*( ) and *popen*( ) functions defined in the System Interfaces volume of IEEE Std 1003.1-200x. If the first line of a file of shell commands starts with the characters `"#!"`, the results are unspecified.
2. The shell breaks the input into tokens: words and operators; see Section 2.3 (on page 2233).
3. The shell parses the input into simple commands (see Section 2.9.1 (on page 2248)) and compound commands (see Section 2.9.4 (on page 2253)).
4. The shell performs various expansions (separately) on different parts of each command, resulting in a list of pathnames and fields to be treated as a command and arguments; see Section 2.6 (on page 2238).
5. The shell performs redirection (see Section 2.7 (on page 2244)) and removes redirection operators and their operands from the parameter list.
6. The shell executes a function (see Section 2.9.5 (on page 2256)), built-in (see Section 2.14 (on page 2266)), executable file, or script, giving the names of the arguments as positional parameters numbered 1 to *n*, and the name of the command (or in the case of a function within a script, the name of the script) as the positional parameter numbered 0 (see Section 2.9.1.1 (on page 2249)).
7. The shell optionally waits for the command to complete and collects the exit status (see Section 2.8.2 (on page 2248)).

2.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to preserve the literal meaning of the special characters in the next paragraph, prevent reserved words from being recognized as such, and prevent parameter expansion and command substitution within here-document processing (see Section 2.7.4 (on page 2246)).

The application shall quote the following characters if they are to represent themselves:

```
|  &  ;  <  >  (  )  $  `  \  "  '  <space>  <tab>  <newline>
```

and the following may need to be quoted under certain circumstances. That is, these characters may be special depending on conditions described elsewhere in this volume of IEEE Std 1003.1-200x:

```
*   ?   [   #   ~   =   %
```

The various quoting mechanisms are the escape character, single-quotes, and double-quotes. The here-document represents another form of quoting; see Section 2.7.4 (on page 2246).

2.2.1 Escape Character (Backslash)

A backslash that is not quoted shall preserve the literal value of the following character, with the exception of a <newline>. If a <newline> follows the backslash, the shell shall interpret this as line continuation. The backslash and <newline>s shall be removed before splitting the input into tokens. Since the escaped <newline> is removed entirely from the input and is not replaced by any white space, it cannot serve as a token separator.

2.2.2 Single-Quotes

Enclosing characters in single-quotes (`''`) shall preserve the literal value of each character within the single-quotes. A single-quote cannot occur within single-quotes.

2.2.3 Double-Quotes

Enclosing characters in double-quotes (`""`) shall preserve the literal value of all characters within the double-quotes, with the exception of the characters dollar sign, backquote, and backslash, as follows:

`$` The dollar sign shall retain its special meaning introducing parameter expansion (see Section 2.6.2 (on page 2239)), a form of command substitution (see Section 2.6.3 (on page 2242)), and arithmetic expansion (see Section 2.6.4 (on page 2243)).

The input characters within the quoted string that are also enclosed between `"$("` and the matching `')'` is not affected by the double-quotes, but rather shall define that command whose output replaces the `"$(...)"` when the word is expanded. The tokenizing rules in Section 2.3 (on page 2233), not including the alias substitutions in Section 2.3.1 (on page 2234), shall be applied recursively to find the matching `')'`.

Within the string of characters from an enclosed `"${"` to the matching `'}'`, an even number of unescaped double-quotes or single-quotes, if any, shall occur. A preceding backslash character shall be used to escape a literal `'{'` or `'}'`. The rule in Section 2.6.2 (on page 2239) shall be used to determine the matching `'}'`.

`` ` `` The backquote shall retain its special meaning introducing the other form of command substitution (see Section 2.6.3 (on page 2242)). The portion of the quoted string from the initial backquote and the characters up to the next backquote that is not preceded by a backslash, having escape characters removed, defines that command whose output replaces `"‘...‘"` when the word is expanded. Either of the following cases produces undefined results:

- A single-quoted or double-quoted string that begins, but does not end, within the `"‘...‘"` sequence
- A `"‘...‘"` sequence that begins, but does not end, within the same double-quoted string

\ The backslash shall retain its special meaning as an escape character (see Section 2.2.1 (on page 2232)) only when followed by one of the following characters when considered special:

`$ ‘ " \ <newline>`

The application shall ensure that a double-quote is preceded by a backslash to be included within double-quotes. The parameter `’@’` has special meaning inside double-quotes and is described in Section 2.5.2 (on page 2235).

2.3 Token Recognition

The shell shall read its input in terms of lines from a file, from a terminal in the case of an interactive shell, or from a string in the case of *sh* **–c** or *system*( ). The input lines can be of unlimited length. These lines shall be parsed using two major modes: ordinary token recognition and processing of here-documents.

When an **io_here** token has been recognized by the grammar (see Section 2.10 (on page 2257)), one or more of the subsequent lines immediately following the next **NEWLINE** token form the body of one or more here-documents and shall be parsed according to the rules of Section 2.7.4 (on page 2246).

When it is not processing an **io_here**, the shell shall break its input into tokens by applying the first applicable rule below to the next character in its input. The token shall be from the current position in the input until a token is delimited according to one of the rules below; the characters forming the token are exactly those in the input, including any quoting characters. If it is indicated that a token is delimited, and no characters have been included in a token, processing shall continue until an actual token is delimited.

1. If the end of input is recognized, the current token shall be delimited. If there is no current token, the end-of-input indicator shall be returned as the token.
2. If the previous character was used as part of an operator and the current character is not quoted and can be used with the current characters to form an operator, it shall be used as part of that (operator) token.
3. If the previous character was used as part of an operator and the current character cannot be used with the current characters to form an operator, the operator containing the previous character shall be delimited.
4. If the current character is backslash, single-quote, or double-quote (`’\\’`, `’\’’`, or `’"’`) and it is not quoted, it shall affect quoting for subsequent characters up to the end of the quoted text. The rules for quoting are as described in Section 2.2 (on page 2232). During token recognition no substitutions shall be actually performed, and the result token shall contain exactly the characters that appear in the input (except for <newline> joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the quote mark and the end of the quoted text. The token shall not be delimited by the end of the quoted field.

5. If the current character is an unquoted `'$'` or `'``'`, the shell shall identify the start of any candidates for parameter expansion (Section 2.6.2 (on page 2239)), command substitution (Section 2.6.3 (on page 2242)), or arithmetic expansion (Section 2.6.4 (on page 2243)) from their introductory unquoted character sequences: `'$'` or `"${"`, `"$("` or `'``'`, and `"$(("`, respectively. The shell shall read sufficient input to determine the end of the unit to be expanded (as explained in the cited sections). While processing the characters, if instances of expansions or quoting are found nested within the substitution, the shell shall recursively process them in the manner specified for the construct that is found. The characters found from the beginning of the substitution to its end, allowing for any recursion necessary to recognize embedded constructs, shall be included unmodified in the result token, including any embedded or enclosing substitution operators or quotes. The token shall not be delimited by the end of the substitution.

6. If the current character is not quoted and can be used as the first character of a new operator, the current token (if any) shall be delimited. The current character shall be used as the beginning of the next (operator) token.

7. If the current character is an unquoted <newline>, the current token shall be delimited.

8. If the current character is an unquoted <blank>, any token containing the previous character is delimited and the current character shall be discarded.

9. If the previous character was part of a word, the current character shall be appended to that word.

10. If the current character is a `'#'`, it and all subsequent characters up to, but excluding, the next <newline> shall be discarded as a comment. The <newline> that ends the line is not considered part of the comment.

11. The current character is used as the start of a new word.

Once a token is delimited, it is categorized as required by the grammar in Section 2.10 (on page 2257).

2.3.1 Alias Substitution

UP XSI The processing of aliases shall be supported on all XSI-conformant systems or if the system supports the User Portability Utilities option (and the rest of this section is not further shaded for these options).

After a token has been delimited, but before applying the grammatical rules in Section 2.10 (on page 2257), a resulting word that is identified to be the command name word of a simple command shall be examined to determine whether it is an unquoted, valid alias name. However, reserved words in correct grammatical context shall not be candidates for alias substitution. A valid alias name (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.10, Alias Name) shall be one that has been defined by the *alias* utility and not subsequently undefined using *unalias*. Implementations also may provide predefined valid aliases that are in effect when the shell is invoked. To prevent infinite loops in recursive aliasing, if the shell is not currently processing an alias of the same name, the word shall be replaced by the value of the alias; otherwise, it shall not be replaced.

If the value of the alias replacing the word ends in a <blank>, the shell shall check the next command word for alias substitution; this process shall continue until a word is found that is not a valid alias or an alias value does not end in a <blank>.

When used as specified by this volume of IEEE Std 1003.1-200x, alias definitions shall not be inherited by separate invocations of the shell or by the utility execution environments invoked by the shell; see Section 2.12 (on page 2263).

2.4 Reserved Words

Reserved words are words that have special meaning to the shell; see Section 2.9 (on page 2248). The following words shall be recognized as reserved words:

!
{
}
case
do
done
elif
else
esac
fi
for
if
in
then
until
while

This recognition shall only occur when none of the characters is quoted and when the word is used as:

- The first word of a command
- The first word following one of the reserved words other than **case**, **for**, or **in**
- The third word in a **case** or **for** command (only **in** is valid in this case)

See the grammar in Section 2.10 (on page 2257).

The following words may be recognized as reserved words on some implementations (when none of the characters are quoted), causing unspecified results:

[[
]]
function
select

Words that are the concatenation of a name and a colon (`':'`) are reserved; their use produces unspecified results.

2.5 Parameters and Variables

A parameter can be denoted by a name, a number, or one of the special characters listed in Section 2.5.2. A variable is a parameter denoted by a name.

A parameter is set if it has an assigned value (null is a valid value). Once a variable is set, it can only be unset by using the *unset* special built-in command.

2.5.1 Positional Parameters

A positional parameter is a parameter denoted by the decimal value represented by one or more digits, other than the single digit 0. The digits denoting the positional parameters shall always be interpreted as a decimal value, even if there is a leading zero. When a positional parameter with more than one digit is specified, the application shall enclose the digits in braces (see Section 2.6.2 (on page 2239)). Positional parameters are initially assigned when the shell is invoked (see *sh*), temporarily replaced when a shell function is invoked (see Section 2.9.5 (on page 2256)), and can be reassigned with the *set* special built-in command.

2.5.2 Special Parameters

Listed below are the special parameters and the values to which they shall expand. Only the values of the special parameters are listed; see Section 2.6 (on page 2238) for a detailed summary of all the stages involved in expanding words.

`@` Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, and where field splitting (see Section 2.6.5 (on page 2243)) is performed, each positional parameter shall expand as a separate field, with the provision that the expansion of the first parameter shall still be joined with the beginning part of the original word (assuming that the expanded parameter was embedded within a word), and the expansion of the last parameter shall still be joined with the last part of the original word. If there are no positional parameters, the expansion of `'@'` shall generate zero fields, even when `'@'` is double-quoted.

* Expands to the positional parameters, starting from one. When the expansion occurs within a double-quoted string (see Section 2.2.3 (on page 2232)), it shall expand to a single field with the value of each parameter separated by the first character of the *IFS* variable, or by a <space> if *IFS* is unset. If *IFS* is set to a null string, this is not equivalent to unsetting it; its first character does not exist, so the parameter values are concatenated.

Expands to the decimal number of positional parameters. The command name (parameter 0) shall not be counted in the number given by `'#'` because it is a special parameter, not a positional parameter.

? Expands to the decimal exit status of the most recent pipeline (see Section 2.9.2 (on page 2250)).

– (Hyphen.) Expands to the current option flags (the single-letter option names concatenated into a string) as specified on invocation by the *set* special built-in command or implicitly by the shell.

$ Expands to the decimal process ID of the invoked shell. In a subshell (see Section 2.12 (on page 2263)), `'$'` shall expand to the same value as that of the current shell.

! Expands to the decimal process ID of the most recent background command (see Section 2.9.3 (on page 2251)) executed from the current shell. (For example, background commands executed from subshells do not affect the value of `"$!"` in the current shell environment.) For a pipeline, the process ID is that of the last command in the pipeline.

0 (Zero.) Expands to the name of the shell or shell script. See *sh* (on page 3048) for a detailed description of how this name is derived.

See the description of the *IFS* variable in Section 2.5.3.

2.5.3 Shell Variables

Variables shall be initialized from the environment (as defined by the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables and the *exec* function in the System Interfaces volume of IEEE Std 1003.1-200x) and can be given new values with variable assignment commands. If a variable is initialized from the environment, it shall be marked for export immediately; see the *export* special built-in. New variables can be defined and initialized with variable assignments, with the *read* or *getopts* utilities, with the *name* parameter in a **for** loop, with the ${*name*=*word*} expansion, or with other mechanisms provided as implementation extensions.

The following variables shall affect the execution of the shell.

UP XSI *ENV* The processing of the *ENV* shell variable shall be supported on all XSI-conformant systems or if the system supports the User Portability Utilities option.

This variable, when and only when an interactive shell is invoked, shall be subjected to parameter expansion (see Section 2.6.2 (on page 2239)) by the shell and the resulting value shall be used as a pathname of a file containing shell commands to execute in the current environment. The file need not be executable. If the expanded value of *ENV* is not an absolute pathname, the results are unspecified. *ENV* shall be ignored if the user's real and effective user IDs or real and effective group IDs are different.

HOME	The pathname of the user's home directory. The contents of *HOME* are used in tilde expansion (see Section 2.6.1 (on page 2239)).
IFS	(Input Field Separators.) A string treated as a list of characters that is used for field splitting and to split lines into fields with the *read* command. If *IFS* is not set, the shell shall behave as if the value of *IFS* is <space>, <tab>, and <newline>; see Section 2.6.5 (on page 2243).
LANG	The default value for the internationalization variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-defined default locale is used. If any of the internationalization variables contains an invalid setting, the utility behaves as if none of the variables had been defined.
LC_ALL	The default value for the *LC_** variables, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.
LC_COLLATE	Determine the behavior of range expressions, equivalence classes, and multi-character collating elements within pattern matching.
LC_CTYPE	Determine the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters), which characters are defined as letters (character class **alpha**) and <blank>s (character class **blank**), and the behavior of character classes within pattern matching. Changing the value of *LC_CTYPE* after the shell has started shall not affect the lexical processing of shell commands in the current shell execution environment or its subshells. Invoking a shell script or performing *exec sh* subjects the new shell to the changes in *LC_CTYPE*.
LC_MESSAGES	Determine the language in which messages should be written.
LINENO	Set by the shell to a decimal number representing the current sequential line number (numbered starting with 1) within a script or function before it executes each command. If the user unsets or resets *LINENO*, the variable may lose its special meaning for the life of the shell. If the shell is not currently executing a script or function, the value of *LINENO* is unspecified. This volume of IEEE Std 1003.1-200x specifies the effects of the variable only for systems supporting the User Portability Utilities option.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	A string formatted as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables, used to effect command interpretation; see Section 2.9.1.1 (on page 2249).
PPID	Set by the shell to the decimal process ID of the process that invoked this shell. In a subshell (see Section 2.12 (on page 2263)), *PPID* shall be set to the same value as that of the parent of the current shell. For example, *echo*$*PPID* and (*echo*$*PPID*) would produce the same value. This volume of IEEE Std 1003.1-200x specifies the effects of the variable only for systems supporting the User Portability Utilities option.
PS1	Each time an interactive shell is ready to read a command, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value shall be `"$ "`. For users who have specific additional implementation-defined privileges, the default may be another, implementation-defined value. The shell shall replace each instance of the

	character `'!'` in *PS1* with the history file number of the next command to be typed. Escaping the `'!'` with another `'!'` (that is, `"!!"`) shall place the literal character `'!'` in the prompt. This volume of IEEE Std 1003.1-200x specifies the effects of the variable only for systems supporting the User Portability Utilities option.
PS2	Each time the user enters a <newline> prior to completing a command line in an interactive shell, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value is `"> "`. This volume of IEEE Std 1003.1-200x specifies the effects of the variable only for systems supporting the User Portability Utilities option.
PS4	When an execution trace (*set* −**x**) is being performed in an interactive shell, before each line in the execution trace, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value is `"+ "`. This volume of IEEE Std 1003.1-200x specifies the effects of the variable only for systems supporting the User Portability Utilities option.
PWD	Set by the shell to be an absolute pathname of the current working directory, containing no components of type symbolic link, no components that are dot, and no components that are dot-dot when the shell is initialized. If an application sets or unsets the value of *PWD*, the behaviors of the *cd* and *pwd* utilities are unspecified.

2.6 Word Expansions

This section describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained in the following sections.

Tilde expansions, parameter expansions, command substitutions, arithmetic expansions, and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple fields from a single word. The single exception to this rule is the expansion of the special parameter `'@'` within double-quotes, as described in Section 2.5.2 (on page 2235).

The order of word expansion shall be as follows:

1. Tilde expansion (see Section 2.6.1 (on page 2239)), parameter expansion (see Section 2.6.2 (on page 2239)), command substitution (see Section 2.6.3 (on page 2242)), and arithmetic expansion (see Section 2.6.4 (on page 2243)) shall be performed, beginning to end. See item 5 in Section 2.3 (on page 2233).
2. Field splitting (see Section 2.6.5 (on page 2243)) shall be performed on the portions of the fields generated by step 1, unless *IFS* is null.
3. Pathname expansion (see Section 2.6.6 (on page 2244)) shall be performed, unless *set* −**f** is in effect.
4. Quote removal (see Section 2.6.7 (on page 2244)) shall always be performed last.

The expansions described in this section shall occur in the same shell environment as that in which the command is executed.

If the complete expansion appropriate for a word results in an empty field, that empty field shall be deleted from the list of fields that form the completely expanded command, unless the original word contained single-quote or double-quote characters.

The `'$'` character is used to introduce parameter expansion, command substitution, or arithmetic evaluation. If an unquoted `'$'` is followed by a character that is either not numeric, the name of one of the special parameters (see Section 2.5.2 (on page 2235)), a valid first character of a variable name, a left curly brace (`'{'`) or a left parenthesis, the result is unspecified.

2.6.1 Tilde Expansion

A *tilde-prefix* consists of an unquoted tilde character at the beginning of a word, followed by all of the characters preceding the first unquoted slash in the word, or all the characters in the word if there is no slash. In an assignment (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.21, Variable Assignment), multiple tilde-prefixes can be used: at the beginning of the word (that is, following the equal sign of the assignment), following any unquoted colon, or both. A tilde-prefix in an assignment is terminated by the first unquoted colon or slash. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible login name from the user database. A portable login name cannot contain characters outside the set given in the description of the *LOGNAME* environment variable in the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.3, Other Environment Variables. If the login name is null (that is, the tilde-prefix contains only the tilde), the tilde-prefix is replaced by the value of the variable *HOME*. If *HOME* is unset, the results are unspecified. Otherwise, the tilde-prefix shall be replaced by a pathname of the initial working directory associated with the login name obtained using the *getpwnam*( ) function as defined in the System Interfaces volume of IEEE Std 1003.1-200x. If the system does not recognize the login name, the results are undefined.

2.6.2 Parameter Expansion

The format for parameter expansion is as follows:

```
${expression}
```

where *expression* consists of all characters until the matching `'}'`. Any `'}'` escaped by a backslash or within a quoted string, and characters in embedded arithmetic expansions, command substitutions, and variable expansions, shall not be examined in determining the matching `'}'`.

The simplest form for parameter expansion is:

```
${parameter}
```

The value, if any, of *parameter* shall be substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when *parameter* is followed by a character that could be interpreted as part of the name. The matching closing brace shall be determined by counting brace levels, skipping over enclosed quoted strings, and command substitutions.

If the parameter name or symbol is not enclosed in braces, the expansion shall use the longest valid name (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.230, Name), whether or not the symbol represented by that name exists.

If a parameter expansion occurs inside double-quotes:

- Pathname expansion shall not be performed on the results of the expansion.
- Field splitting shall not be performed on the results of the expansion, with the exception of `'@'`; see Section 2.5.2 (on page 2235).

In addition, a parameter expansion can be modified by using one of the following formats. In each case that a value of *word* is needed (based on the state of *parameter*, as described below), *word* shall be subjected to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. If *word* is not needed, it shall not be expanded. The `'}'` character that delimits the following parameter expansion modifications shall be determined as described previously in this section and in Section 2.2.3 (on page 2232). (For example, ${**foo-bar**}**xyz**} would result in the expansion of **foo** followed by the string **xyz**} if **foo** is set, else the string `"barxyz}"`).

${*parameter*:−*word*} **Use Default Values**. If *parameter* is unset or null, the expansion of *word* shall be substituted; otherwise, the value of *parameter* shall be substituted.

${*parameter*:=*word*} **Assign Default Values**. If *parameter* is unset or null, the expansion of *word* shall be assigned to *parameter*. In all cases, the final value of *parameter* shall be substituted. Only variables, not positional parameters or special parameters, can be assigned in this way.

${*parameter*:?**[***word***]**} **Indicate Error if Null or Unset**. If *parameter* is unset or null, the expansion of *word* (or a message indicating it is unset if *word* is omitted) shall be written to standard error and the shell exits with a non-zero exit status. Otherwise, the value of *parameter* shall be substituted. An interactive shell need not exit.

${*parameter*:+*word*} **Use Alternative Value**. If *parameter* is unset or null, null shall be substituted; otherwise, the expansion of *word* shall be substituted.

In the parameter expansions shown previously, use of the colon in the format shall result in a test for a parameter that is unset or null; omission of the colon shall result in a test for a parameter that is only unset. The following table summarizes the effect of the colon:

	parameter **Set and Not Null**	*parameter* **Set But Null**	*parameter* **Unset**
${*parameter***:−***word***}**	substitute *parameter*	substitute *word*	substitute *word*
${*parameter***−***word***}**	substitute *parameter*	substitute null	substitute *word*
${*parameter***:=***word***}**	substitute *parameter*	assign *word*	assign *word*
${*parameter***=***word***}**	substitute *parameter*	substitute *parameter*	assign null
${*parameter***:?***word***}**	substitute *parameter*	error, exit	error, exit
${*parameter***?***word***}**	substitute *parameter*	substitute null	error, exit
${*parameter***:+***word***}**	substitute *word*	substitute null	substitute null
${*parameter***+***word***}**	substitute *word*	substitute *word*	substitute null

In all cases shown with ‘‘substitute’’, the expression is replaced with the value shown. In all cases shown with ‘‘assign’’, *parameter* is assigned that value, which also replaces the expression.

${#*parameter*} **String Length**. The length in characters of the value of *parameter* shall be substituted. If *parameter* is `'*'` or `'@'`, the result of the expansion is unspecified.

The following four varieties of parameter expansion provide for substring processing. In each case, pattern matching notation (see Section 2.13 (on page 2264)), rather than regular expression notation, shall be used to evaluate the patterns. If *parameter* is `'*'` or `'@'`, the result of the expansion is unspecified. Enclosing the full parameter expansion string in double-quotes shall not cause the following four varieties of pattern characters to be quoted, whereas quoting characters within the braces shall have this effect.

${*parameter*%*word*} **Remove Smallest Suffix Pattern**. The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the smallest portion of the suffix matched by the *pattern* deleted.

${*parameter*%%*word*} **Remove Largest Suffix Pattern**. The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the largest portion of the suffix matched by the *pattern* deleted.

${*parameter*#*word*} **Remove Smallest Prefix Pattern**. The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the smallest portion of the prefix matched by the *pattern* deleted.

${*parameter*##*word*} **Remove Largest Prefix Pattern**. The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the largest portion of the prefix matched by the *pattern* deleted.

Examples

${*parameter*:−*word*}

In this example, *ls* is executed only if *x* is null or unset. (The $(*ls*) command substitution notation is explained in Section 2.6.3 (on page 2242).)

```
${x:−$(ls)}
```

${*parameter*:=*word*}

```
unset X
echo ${X:=abc}
abc
```

${*parameter*:?*word*}

```
unset posix
echo ${posix:?}
sh: posix: parameter null or not set
```

${*parameter*:+*word*}

```
set a b c
echo ${3:+posix}
posix
```

${#*parameter*}

```
HOME=/usr/posix
echo ${#HOME}
10
```

${*parameter*%*word*}

```
x=file.c
echo ${x%.c}.o
file.o
```

${*parameter*%%*word*}

```
x=posix/src/std
echo ${x%%/*}
posix
```

${*parameter*#*word*}

```
x=$HOME/src/cmd
echo ${x#$HOME}
/src/cmd
```

${*parameter*##*word*}

```
x=/one/two/three
```

```
echo ${x##*/}
three
```

The double-quoting of patterns is different depending on where the double-quotes are placed:

`"${x#*}"` The asterisk is a pattern character.

`${x#"*"}` The literal asterisk is quoted and not special.

2.6.3 Command Substitution

Command substitution allows the output of a command to be substituted in place of the command name itself. Command substitution shall occur when the command is enclosed as follows:

```
$(command)
```

or (backquoted version):

```
‘command‘
```

The shell shall expand the command substitution by executing *command* in a subshell environment (see Section 2.12 (on page 2263)) and replacing the command substitution (the text of *command* plus the enclosing `"$()"` or backquotes) with the standard output of the command, removing sequences of one or more <newline>s at the end of the substitution. Embedded <newline>s before the end of the output shall not be removed; however, they may be treated as field delimiters and eliminated during field splitting, depending on the value of *IFS* and quoting that is in effect.

Within the backquoted style of command substitution, backslash shall retain its literal meaning, except when followed by: `’$’`, `’‘’`, or `’\\’` (dollar sign, backquote, backslash). The search for the matching backquote shall be satisfied by the first backquote found without a preceding backslash; during this search, if a non-escaped backquote is encountered within a shell comment, a here-document, an embedded command substitution of the $(*command*) form, or a quoted string, undefined results occur. A single-quoted or double-quoted string that begins, but does not end, within the `"‘...‘"` sequence produces undefined results.

With the $(*command*) form, all characters following the open parenthesis to the matching closing parenthesis constitute the *command*. Any valid shell script can be used for *command*, except:

- A script consisting solely of redirections produces unspecified results
- See the restriction on single subshells described below

The results of command substitution shall not be processed for further tilde expansion, parameter expansion, command substitution, or arithmetic expansion. If a command substitution occurs inside double-quotes, it shall not be performed on the results of the substitution.

Command substitution can be nested. To specify nesting within the backquoted version, the application shall precede the inner backquotes with backslashes, for example:

```
\‘command\‘
```

If the command substitution consists of a single subshell, such as:

```
$( (command) )
```

a conforming application shall separate the `"$("` and `’(’` into two tokens (that is, separate them with white space). This is required to avoid any ambiguities with arithmetic expansion.

2.6.4 Arithmetic Expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion shall be as follows:

```
$((expression))
```

The expression shall be treated as if it were in double-quotes, except that a double-quote inside the expression is not treated specially. The shell shall expand all tokens in the expression for parameter expansion, command substitution, and quote removal.

Next, the shell shall treat this as an arithmetic expression and substitute the value of the expression. The arithmetic expression shall be processed according to the rules given in Section 1.7.2.1 (on page 2207), with the following exceptions:

- Only signed long integer arithmetic is required.
- Only the decimal-constant, octal-constant, and hexadecimal-constant constants specified in the ISO C standard, Subclause 6.4.4.1 are required to be recognized as constants.
- The *sizeof*( ) operator and the prefix and postfix "++" and "−−" operators are not required.
- Selection, iteration, and jump statements are not supported.

As an extension, the shell may recognize arithmetic expressions beyond those listed. The shell may use a signed integer type with a rank larger than the rank of **signed long**. The shell may use a real-floating type instead of **signed long** as long as it does not affect the results in cases where there is no overflow. If the expression is invalid, the expansion fails and the shell shall write a message to standard error indicating the failure.

Examples

A simple example using arithmetic expansion:

```
# repeat a command 100 times
x=100
while [ $x −gt 0 ]
do
    command
    x=$(($x−1))
done
```

2.6.5 Field Splitting

After parameter expansion (Section 2.6.2 (on page 2239)), command substitution (Section 2.6.3 (on page 2242)), and arithmetic expansion (Section 2.6.4), the shell shall scan the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

The shell shall treat each character of the *IFS* as a delimiter and use the delimiters to split the results of parameter expansion and command substitution into fields.

1. If the value of *IFS* is a <space>, <tab>, and <newline>, or if it is unset, any sequence of <space>s, <tab>s, or <newline>s at the beginning or end of the input shall be ignored and any sequence of those characters within the input shall delimit a field. For example, the input:

   ```
   <newline><space><tab>foo<tab><tab>bar<space>
   ```

yields two fields, **foo** and **bar**.

2. If the value of *IFS* is null, no field splitting shall be performed.

3. Otherwise, the following rules shall be applied in sequence. The term ''*IFS* white space'' is used to mean any sequence (zero or more instances) of white space characters that are in the *IFS* value (for example, if *IFS* contains <space>/<comma>/<tab>, any sequence of <space>s and <tab>s is considered *IFS* white space).

   a. *IFS* white space shall be ignored at the beginning and end of the input.

   b. Each occurrence in the input of an *IFS* character that is not *IFS* white space, along with any adjacent *IFS* white space, shall delimit a field, as described previously.

   c. Non-zero-length *IFS* white space shall delimit a field.

2.6.6 Pathname Expansion

After field splitting, if *set* **−f** is not in effect, each field in the resulting command line shall be expanded using the algorithm described in Section 2.13 (on page 2264), qualified by the rules in Section 2.13.3 (on page 2265).

2.6.7 Quote Removal

The quote characters: `'\'`, `'\''`, and `'"'` (backslash, single-quote, double-quote) that were present in the original word shall be removed unless they have themselves been quoted.

2.7 Redirection

Redirection is used to open and close files for the current shell execution environment (see Section 2.12 (on page 2263)) or for any command. *Redirection operators* can be used with numbers representing file descriptors (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.165, File Descriptor) as described below.

The overall format used for redirection is:

```
[n]redir-op word
```

The number *n* is an optional decimal number designating the file descriptor number; the application shall ensure it is delimited from any preceding text and immediately precede the redirection operator *redir-op*. If *n* is quoted, the number shall not be recognized as part of the redirection expression. For example:

```
echo \2>a
```

writes the character 2 into file **a**. If any part of *redir-op* is quoted, no redirection expression is recognized. For example:

```
echo 2\>a
```

writes the characters 2>*a* to standard output. The optional number, redirection operator, and *word* shall not appear in the arguments provided to the command to be executed (if any).

Open files are represented by decimal numbers starting with zero. The largest possible value is implementation-defined; however, all implementations shall support at least 0 to 9, inclusive, for use by the application. These numbers are called *file descriptors*. The values 0, 1, and 2 have special meaning and conventional uses and are implied by certain redirection operations; they are referred to as *standard input*, *standard output*, and *standard error*, respectively. Programs usually take their input from standard input, and write output on standard output. Error messages are usually written on standard error. The redirection operators can be preceded by one or more digits (with no intervening <blank>s allowed) to designate the file descriptor number.

If the redirection operator is `"<<"` or `"<<–"`, the word that follows the redirection operator shall be subjected to quote removal; it is unspecified whether any of the other expansions occur. For the other redirection operators, the word that follows the redirection operator shall be subjected to tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion shall not be performed on the word by a non-interactive shell; an interactive shell may perform it, but shall do so only when the expansion would result in one word.

If more than one redirection operator is specified with a command, the order of evaluation is from beginning to end.

A failure to open or create a file shall cause a redirection to fail.

2.7.1 Redirecting Input

Input redirection shall cause the file whose name results from the expansion of *word* to be opened for reading on the designated file descriptor, or standard input if the file descriptor is not specified.

The general format for redirecting input is:

```
[n]<word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard input (file descriptor 0).

2.7.2 Redirecting Output

The two general formats for redirecting output are:

```
[n]>word
[n]>|word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard output (file descriptor 1).

Output redirection using the `'>'` format shall fail if the *noclobber* option is set (see the description of *set* **–C**) and the file named by the expansion of *word* exists and is a regular file. Otherwise, redirection using the `'>'` or `">|"` formats shall cause the file whose name results from the expansion of *word* to be created and opened for output on the designated file descriptor, or standard output if none is specified. If the file does not exist, it shall be created; otherwise, it shall be truncated to be an empty file after being opened.

2.7.3 Appending Redirected Output

Appended output redirection shall cause the file whose name results from the expansion of word to be opened for output on the designated file descriptor. The file is opened as if the *open*( ) function as defined in the System Interfaces volume of IEEE Std 1003.1-200x was called with the O_APPEND flag. If the file does not exist, it shall be created.

The general format for appending redirected output is as follows:

```
[n]>>word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection refers to standard output (file descriptor 1).

2.7.4 Here-Document

The redirection operators `"<<"` and `"<<−"` both allow redirection of lines contained in a shell input file, known as a *here-document*, to the input of a command.

The here-document shall be treated as a single word that begins after the next <newline> and continues until there is a line containing only the delimiter and a <newline>, with no <blank>s in between. Then the next here-document starts, if there is one. The format is as follows:

```
[n]<<word
    here-document
delimiter
```

where the optional *n* represents the file descriptor number. If the number is omitted, the here-document refers to standard input (file descriptor 0).

If any character in *word* is quoted, the delimiter shall be formed by performing quote removal on *word*, and the here-document lines shall not be expanded. Otherwise, the delimiter shall be the *word* itself.

If no characters in *word* are quoted, all lines of the here-document shall be expanded for parameter expansion, command substitution, and arithmetic expansion. In this case, the backslash in the input behaves as the backslash inside double-quotes (see Section 2.2.3 (on page 2232)). However, the double-quote character (`'"'`) shall not be treated specially within a here-document, except when the double-quote appears within `"$()"`, ``"``"``, or `"${}"`.

If the redirection symbol is `"<<−"`, all leading tab characters shall be stripped from input lines and the line containing the trailing delimiter. If more than one `"<<"` or `"<<−"` operator is specified on a line, the here-document associated with the first operator shall be supplied first by the application and shall be read first by the shell.

Examples

An example of a here-document follows:

```
cat <<eof1; cat <<eof2
Hi,
eof1
Helene.
eof2
```

2.7.5 Duplicating an Input File Descriptor

The redirection operator:

```
[n]<&word
```

shall duplicate one input file descriptor from another, or shall close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard input if *n* is not specified, shall be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for input, a redirection error shall result; see Section 2.8.1 (on page 2247). If *word* evaluates to `'−'`, file descriptor *n*, or standard input if *n* is not specified, shall be closed. If *word* evaluates to something else, the behavior is unspecified.

2.7.6 Duplicating an Output File Descriptor

The redirection operator:

```
[n]>&word
```

shall duplicate one output file descriptor from another, or shall close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard output if *n* is not specified, shall be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for output, a redirection error shall result; see Section 2.8.1. If *word* evaluates to `'−'`, file descriptor *n*, or standard output if *n* is not specified, is closed. If *word* evaluates to something else, the behavior is unspecified.

2.7.7 Open File Descriptors for Reading and Writing

The redirection operator:

```
[n]<>word
```

shall cause the file whose name is the expansion of *word* to be opened for both reading and writing on the file descriptor denoted by *n*, or standard input if *n* is not specified. If the file does not exist, it shall be created.

2.8 Exit Status and Errors

2.8.1 Consequences of Shell Errors

For a non-interactive shell, an error condition encountered by a special built-in (see Section 2.14 (on page 2266)) or other type of utility shall cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

Error	Special Built-In	Other Utilities
Shell language syntax error	Shall exit	Shall exit
Utility syntax error (option or operand error)	Shall exit	Shall not exit
Redirection error	Shall exit	Shall not exit
Variable assignment error	Shall exit	Shall not exit
Expansion error	Shall exit	Shall exit
Command not found	N/A	May exit
Dot script not found	Shall exit	N/A

An expansion error is one that occurs when the shell expansions defined in Section 2.6 (on page 2238) are carried out (for example, `"${x!y}"`, because `'!'` is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenization, rather than during expansion.

If any of the errors shown as ‘‘shall exit’’ or ‘‘(may) exit’’ occur in a subshell, the subshell shall (respectively may) exit with a non-zero status, but the script containing the subshell shall not exit because of the error.

In all of the cases shown in the table, an interactive shell shall write a diagnostic message to standard error without exiting.

2.8.2 Exit Status for Commands

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities is documented in this section. The exit status of the standard utilities is documented in their respective sections.

If a command is not found, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

If a command fails during word expansion or redirection, its exit status shall be greater than zero.

Internally, for purposes of deciding whether a command exits with a non-zero exit status, the shell shall recognize the entire status value retrieved for the command by the equivalent of the *wait*( ) function WEXITSTATUS macro (as defined in the System Interfaces volume of IEEE Std 1003.1-200x). When reporting the exit status with the special parameter `'?'`, the shell shall report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal shall be reported as greater than 128.

2.9 Shell Commands

This section describes the basic structure of shell commands. The following command descriptions each describe a format of the command that is only used to aid the reader in recognizing the command type, and does not formally represent the syntax. Each description discusses the semantics of the command; for a formal definition of the command language, consult Section 2.10 (on page 2257).

A *command* is one of the following:

- *Simple command* (see Section 2.9.1)
- *Pipeline* (see Section 2.9.2 (on page 2250))
- *List* or *compound-list* (see Section 2.9.3 (on page 2251))
- *Compound command* (see Section 2.9.4 (on page 2253))
- *Function definition* (see Section 2.9.5 (on page 2256))

Unless otherwise stated, the exit status of a command shall be that of the last simple command executed by the command. There shall be no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG_MAX}, and so on).

2.9.1 Simple Commands

A *simple command* is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by words and redirections, terminated by a control operator.

When a given simple command is required to be executed (that is, when any conditional construct such as an AND-OR list or a **case** statement has not bypassed the simple command), the following expansions, assignments, and redirections shall all be performed from the beginning of the command text to the end:

1. The words that are recognized as variable assignments or redirections according to Section 2.10.2 (on page 2257) are saved for processing in steps 3 and 4.
2. The words that are not variable assignments or redirections shall be expanded. If any fields remain following their expansion, the first field shall be considered the command name

and remaining fields are the arguments for the command.

3. Redirections shall be performed as described in Section 2.7 (on page 2244).
4. Each variable assignment shall be expanded for tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal prior to assigning the value.

In the preceding list, the order of steps 3 and 4 may be reversed for the processing of special built-in utilities; see Section 2.14 (on page 2266).

If no command name results, variable assignments shall affect the current execution environment. Otherwise, the variable assignments shall be exported for the execution environment of the command and shall not affect the current execution environment (except for special built-ins). If any of the variable assignments attempt to assign a value to a read-only variable, a variable assignment error shall occur. See Section 2.8.1 (on page 2247) for the consequences of these errors.

If there is no command name, any redirections shall be performed in a subshell environment; it is unspecified whether this subshell environment is the same one as that used for a command substitution within the command. (To affect the current execution environment, see the *exec* (on page 2277) special built-in.) If any of the redirections performed in the current shell execution environment fail, the command shall immediately fail with an exit status greater than zero, and the shell shall write an error message indicating the failure. See Section 2.8.1 (on page 2247) for the consequences of these failures on interactive and non-interactive shells.

If there is a command name, execution shall continue as described in Section 2.9.1.1. If there is no command name, but the command contained a command substitution, the command shall complete with the exit status of the last command substitution performed. Otherwise, the command shall complete with a zero exit status.

2.9.1.1 Command Search and Execution

If a simple command results in a command name and an optional list of arguments, the following actions shall be performed:

1. If the command name does not contain any slashes, the first successful step in the following sequence shall occur:
    a. If the command name matches the name of a special built-in utility, that special built-in utility shall be invoked.

    b. If the command name matches the name of a function known to this shell, the function shall be invoked as described in Section 2.9.5 (on page 2256). If the implementation has provided a standard utility in the form of a function, it shall not be recognized at this point. It shall be invoked in conjunction with the path search in step 1d.

    c. If the command name matches the name of a utility listed in the following table, that utility shall be invoked.

    alias
    bg
    cd
    command
    false
    fc
    fg
    getopts
    jobs
    kill
    newgrp
    read
    true
    umask
    unalias
    wait

    d. Otherwise, the command shall be searched for using the *PATH* environment variable as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables:

i. If the search is successful:

a. If the system has implemented the utility as a regular built-in or as a shell function, it shall be invoked at this point in the path search.

b. Otherwise, the shell executes the utility in a separate utility environment (see Section 2.12 (on page 2263)) with actions equivalent to calling the *execve*( ) function as defined in the System Interfaces volume of IEEE Std 1003.1-200x with the *path* argument set to the pathname resulting from the search, *arg*0 set to the command name, and the remaining arguments set to the operands, if any.

If the *execve*( ) function fails due to an error equivalent to the [ENOEXEC] error defined in the System Interfaces volume of IEEE Std 1003.1-200x, the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, with any remaining arguments passed to the new shell. If the executable file is not a text file, the shell may bypass this command execution. In this case, it shall write an error message, and shall return an exit status of 126.

Once a utility has been searched for and found (either as a result of this specific search or as part of an unspecified shell start-up activity), an implementation may remember its location and need not search for the utility again unless the *PATH* variable has been the subject of an assignment. If the remembered location fails for a subsequent invocation, the shell shall repeat the search to find the new location for the utility, if any.

ii. If the search is unsuccessful, the command shall fail with an exit status of 127 and the shell shall write an error message.

2. If the command name contains at least one slash, the shell shall execute the utility in a separate utility environment with actions equivalent to calling the *execve*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x with the *path* and *arg*0 arguments set to the command name, and the remaining arguments set to the operands, if any.

   If the *execve*( ) function fails due to an error equivalent to the [ENOEXEC] error, the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, with any remaining arguments passed to the new shell. If the executable file is not a text file, the shell may bypass this command execution. In this case, it shall write an error message and shall return an exit status of 126.

2.9.2 Pipelines

A *pipeline* is a sequence of one or more commands separated by the control operator `'|'`. The standard output of all but the last command shall be connected to the standard input of the next command.

The format for a pipeline is:

```
[!] command1 [ | command2 ...]
```

The standard output of *command1* shall be connected to the standard input of *command2*. The standard input, standard output, or both of a command shall be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command (see Section 2.7 (on page 2244)).

If the pipeline is not in the background (see Section 2.9.3.1 (on page 2252)), the shell shall wait for the last command specified in the pipeline to complete, and may also wait for all commands to complete.

Exit Status

If the reserved word `!` does not precede the pipeline, the exit status shall be the exit status of the last command specified in the pipeline. Otherwise, the exit status shall be the logical NOT of the exit status of the last command. That is, if the last command returns zero, the exit status shall be 1; if the last command returns greater than zero, the exit status shall be zero.

2.9.3 Lists

An *AND-OR list* is a sequence of one or more pipelines separated by the operators `"&&"` and `"||"`.

A *list* is a sequence of one or more AND-OR lists separated by the operators `';'` and `'&'` and optionally terminated by `';'`, `'&'`, or <newline>.

The operators `"&&"` and `"||"` shall have equal precedence and shall be evaluated with left associativity. For example, both of the following commands write solely **bar** to standard output:

```
false && echo foo || echo bar
true || echo foo && echo bar
```

A `';'` or <newline> terminator shall cause the preceding AND-OR list to be executed sequentially; an `'&'` shall cause asynchronous execution of the preceding AND-OR list.

The term *compound-list* is derived from the grammar in Section 2.10 (on page 2257); it is equivalent to a sequence of *lists*, separated by <newline>s, that can be preceded or followed by an arbitrary number of <newline>s.

Examples

The following is an example that illustrates <newline>s in compound-lists:

```
while
    # a couple of <newline>s

    # a list
    date && who || ls; cat file
    # a couple of <newline>s

    # another list
    wc file > output & true

do
    # 2 lists
    ls
    cat file
done
```

2.9.3.1 *Asynchronous Lists*

If a command is terminated by the control operator ampersand (`'&'`), the shell shall execute the command asynchronously in a subshell. This means that the shell shall not wait for the command to finish before executing the next command.

The format for running a command in the background is:

```
command1 & [command2 & ... ]
```

The standard input for an asynchronous list, before any explicit redirections are performed, shall be considered to be assigned to a file that has the same properties as **/dev/null**. If it is an interactive shell, this need not happen. In all cases, explicit redirection of standard input shall override this activity.

When an element of an asynchronous list (the portion of the list ended by an ampersand, such as *command1*, above) is started by the shell, the process ID of the last command in the asynchronous list element shall become known in the current shell execution environment; see Section 2.12 (on page 2263). This process ID shall remain known until:

1. The command terminates and the application waits for the process ID.
2. Another asynchronous list invoked before `"$!"` (corresponding to the previous asynchronous list) is expanded in the current execution environment.

The implementation need not retain more than the {CHILD_MAX} most recent entries in its list of known process IDs in the current shell execution environment.

Exit Status

The exit status of an asynchronous list shall be zero.

2.9.3.2 *Sequential Lists*

Commands that are separated by a semicolon (`';'`) shall be executed sequentially.

The format for executing commands sequentially shall be:

```
command1 [; command2] ...
```

Each command shall be expanded and executed in the order specified.

Exit Status

The exit status of a sequential list shall be the exit status of the last command in the list.

2.9.3.3 *AND Lists*

The control operator `"&&"` denotes an AND list. The format shall be:

```
command1 [ && command2] ...
```

First *command1* shall be executed. If its exit status is zero, *command2* shall be executed, and so on, until a command has a non-zero exit status or there are no more commands left to execute. The commands are expanded only if they are executed.

Exit Status

The exit status of an AND list shall be the exit status of the last command that is executed in the list.

2.9.3.4 *OR Lists*

The control operator `"||"` denotes an OR List. The format shall be:

```
command1 [ || command2] ...
```

First, *command1* shall be executed. If its exit status is non-zero, *command2* shall be executed, and so on, until a command has a zero exit status or there are no more commands left to execute.

Exit Status

The exit status of an OR list shall be the exit status of the last command that is executed in the list.

2.9.4 Compound Commands

The shell has several programming constructs that are *compound commands*, which provide control flow for commands. Each of these compound commands has a reserved word or control operator at the beginning, and a corresponding terminator reserved word or operator at the end. In addition, each can be followed by redirections on the same line as the terminator. Each redirection shall apply to all the commands within the compound command that do not explicitly override that redirection.

2.9.4.1 *Grouping Commands*

The format for grouping commands is as follows:

(*compound-list*)	Execute *compound-list* in a subshell environment; see Section 2.12 (on page 2263). Variable assignments and built-in commands that affect the environment shall not remain in effect after the list finishes.
{ *compound-list*;}	Execute *compound-list* in the current process environment. The semicolon shown here is an example of a control operator delimiting the **}** reserved word. Other delimiters are possible, as shown in Section 2.10 (on page 2257); a <newline> is frequently used.

Exit Status

The exit status of a grouping command shall be the exit status of *list*.

2.9.4.2 *For Loop*

The **for** loop shall execute a sequence of commands for each member in a list of *items*. The **for** loop requires that the reserved words **do** and **done** be used to delimit the sequence of commands.

The format for the **for** loop is as follows:

```
for name [ in [word ... ]]
do
    compound-list
done
```

First, the list of words following **in** shall be expanded to generate a list of items. Then, the variable *name* shall be set to each item, in turn, and the *compound-list* executed each time. If no items result from the expansion, the *compound-list* shall not be executed. Omitting:

```
in word ...
```

shall be equivalent to:

```
in "$@"
```

Exit Status

The exit status of a **for** command shall be the exit status of the last command that executes. If there are no items, the exit status shall be zero.

2.9.4.3 *Case Conditional Construct*

The conditional construct **case** shall execute the *compound-list* corresponding to the first one of several *patterns* (see Section 2.13 (on page 2264)) that is matched by the string resulting from the tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal of the given word. The reserved word **in** shall denote the beginning of the patterns to be matched. Multiple patterns with the same *compound-list* shall be delimited by the `'|'` symbol. The control operator `')'` terminates a list of patterns corresponding to a given action. The *compound-list* for each list of patterns, with the possible exception of the last, shall be terminated with `";;"`. The **case** construct terminates with the reserved word **esac** (**case** reversed).

The format for the **case** construct is as follows:

```
case word in
    [(]pattern1) compound-list;;
    [[(]pattern[ | pattern] ... ) compound-list;;] ...
    [[(]pattern[ | pattern] ... ) compound-list]
esac
```

The `";;"` is optional for the last *compound-list*.

In order from the beginning to the end of the **case** statement, each *pattern* that labels a *compound-list* shall be subjected to tilde expansion, parameter expansion, command substitution, and arithmetic expansion, and the result of these expansions shall be compared against the expansion of *word*, according to the rules described in Section 2.13 (on page 2264) (which also describes the effect of quoting parts of the pattern). After the first match, no more patterns shall be expanded, and the *compound-list* shall be executed. The order of expansion and comparison of multiple *pattern*s that label a *compound-list* statement is unspecified.

Exit Status

The exit status of **case** shall be zero if no patterns are matched. Otherwise, the exit status shall be the exit status of the last command executed in the *compound-list*.

2.9.4.4 *If Conditional Construct*

The **if** command shall execute a *compound-list* and use its exit status to determine whether to execute another *compound-list*.

The format for the **if** construct is as follows:

```
if compound-list
then
    compound-list
[elif compound-list
then
    compound-list] ...
[else
    compound-list]
```

The **if** *compound-list* shall be executed; if its exit status is zero, the **then** *compound-list* shall be executed and the command shall complete. Otherwise, each **elif** *compound-list* shall be executed, in turn, and if its exit status is zero, the **then** *compound-list* shall be executed and the command shall complete. Otherwise, the **else** *compound-list* shall be executed.

Exit Status

The exit status of the **if** command shall be the exit status of the **then** or **else** *compound-list* that was executed, or zero, if none was executed.

2.9.4.5 *While Loop*

The **while** loop shall continuously execute one *compound-list* as long as another *compound-list* has a zero exit status.

The format of the **while** loop is as follows:

```
while compound-list-1
do
    compound-list-2
done
```

The *compound-list-1* shall be executed, and if it has a non-zero exit status, the **while** command shall complete. Otherwise, the *compound-list-2* shall be executed, and the process shall repeat.

Exit Status

The exit status of the **while** loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

2.9.4.6 *Until Loop*

The **until** loop shall continuously execute one *compound-list* as long as another *compound-list* has a non-zero exit status.

The format of the **until** loop is as follows:

```
until compound-list-1
do
    compound-list-2
done
```

The *compound-list-1* shall be executed, and if it has a zero exit status, the **until** command completes. Otherwise, the *compound-list-2* shall be executed, and the process repeats.

Exit Status

The exit status of the **until** loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

2.9.5 Function Definition Command

A function is a user-defined name that is used as a simple command to call a compound command with new positional parameters. A function is defined with a *function definition command*.

The format of a function definition command is as follows:

```
fname() compound-command[io-redirect ...]
```

The function is named *fname*; the application shall ensure that it is a name (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.230, Name). An implementation may allow other characters in a function name as an extension. The implementation shall maintain separate name spaces for functions and variables.

The argument *compound-command* represents a compound command, as described in Section 2.9.4 (on page 2253).

When the function is declared, none of the expansions in Section 2.6 (on page 2238) shall be performed on the text in *compound-command* or *io-redirect*; all expansions shall be performed as normal each time the function is called. Similarly, the optional *io-redirect* redirections and any variable assignments within *compound-command* shall be performed during the execution of the function itself, not the function definition. See Section 2.8.1 (on page 2247) for the consequences of failures of these operations on interactive and non-interactive shells.

When a function is executed, it shall have the syntax-error and variable-assignment properties described for special built-in utilities in the enumerated list at the beginning of Section 2.14 (on page 2266).

The *compound-command* shall be executed whenever the function name is specified as the name of a simple command (see Section 2.9.1.1 (on page 2249)). The operands to the command temporarily shall become the positional parameters during the execution of the *compound-command*; the special parameter `'#'` also shall be changed to reflect the number of operands. The special parameter 0 shall be unchanged. When the function completes, the values of the positional parameters and the special parameter `'#'` shall be restored to the values they had before the function was executed. If the special built-in *return* is executed in the *compound-command*, the function completes and execution shall resume with the next command after the function call.

Exit Status

The exit status of a function definition shall be zero if the function was declared successfully; otherwise, it shall be greater than zero. The exit status of a function invocation shall be the exit status of the last command executed by the function.

2.10 Shell Grammar

The following grammar defines the Shell Command Language. This formal syntax shall take precedence over the preceding text syntax description.

2.10.1 Shell Grammar Lexical Conventions

The input language to the shell must be first recognized at the character level. The resulting tokens shall be classified by their immediate context according to the following rules (applied in order). These rules shall be used to determine what a ‘‘token’’ is that is subject to parsing at the token level. The rules for token recognition in Section 2.3 (on page 2233) shall apply.

1. A <newline> shall be returned as the token identifier **NEWLINE**.
2. If the token is an operator, the token identifier for that operator shall result.
3. If the string consists solely of digits and the delimiter character is one of `'<'` or `'>'`, the token identifier **IO_NUMBER** shall be returned.
4. Otherwise, the token identifier **TOKEN** results.

Further distinction on **TOKEN** is context-dependent. It may be that the same **TOKEN** yields **WORD**, a **NAME**, an **ASSIGNMENT**, or one of the reserved words below, dependent upon the context. Some of the productions in the grammar below are annotated with a rule number from the following list. When a **TOKEN** is seen where one of those annotated productions could be used to reduce the symbol, the applicable rule shall be applied to convert the token identifier type of the **TOKEN** to a token identifier acceptable at that point in the grammar. The reduction shall then proceed based upon the token identifier type yielded by the rule applied. When more than one rule applies, the highest numbered rule shall apply (which in turn may refer to another rule). (Note that except in rule 7, the presence of an `'='` in the token has no effect.)

The **WORD** tokens shall have the word expansion rules applied to them immediately before the associated command is executed, not at the time the command is parsed.

2.10.2 Shell Grammar Rules

1. [Command Name]

   When the **TOKEN** is exactly a reserved word, the token identifier for that reserved word shall result. Otherwise, the token **WORD** shall be returned. Also, if the parser is in any state where only a reserved word could be the next correct token, proceed as above.

   Note: Because at this point quote marks are retained in the token, quoted strings cannot be recognized as reserved words. This rule also implies that reserved words are not recognized except in certain positions in the input, such as after a <newline> or semicolon; the grammar presumes that if the reserved word is intended, it is properly delimited by the user, and does not attempt to reflect that requirement directly. Also note that line joining is done before tokenization, as described in Section 2.2.1 (on page 2232), so escaped <newline>s are already removed at this point.

   Rule 1 is not directly referenced in the grammar, but is referred to by other rules, or applies globally.

2. [Redirection to or from filename]

   The expansions specified in Section 2.7 (on page 2244) shall occur. As specified there, exactly one field can result (or the result is unspecified), and there are additional requirements on pathname expansion.

3. [Redirection from here-document]

   Quote removal shall be applied to the word to determine the delimiter that is used to find the end of the here-document that begins after the next <newline>.

4. [Case statement termination]

   When the **TOKEN** is exactly the reserved word **esac**, the token identifier for **esac** shall result. Otherwise, the token **WORD** shall be returned.

5. [**NAME** in **for**]

   When the **TOKEN** meets the requirements for a name (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.230, Name), the token identifier **NAME** shall result. Otherwise, the token **WORD** shall be returned.

6. [Third word of **for** and **case**]

   When the **TOKEN** is exactly the reserved word **in**, the token identifier for **in** shall result. Otherwise, the token **WORD** shall be returned. (As indicated in the grammar, a *linebreak* precedes the token **in**. If <newline>s are present at the indicated location, it is the token after them that is treated in this fashion.)

7. [Assignment preceding command name]

   a. [When the first word]

      If the **TOKEN** does not contain the character `'='`, rule 1 is applied. Otherwise, 7b shall be applied.

   b. [Not the first word]

      If the **TOKEN** contains the equal sign character:

      — If it begins with `'='`, the token **WORD** shall be returned.

      — If all the characters preceding `'='` form a valid name (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.230, Name), the token **ASSIGNMENT_WORD** shall be returned. (Quoted characters cannot participate in forming a valid name.)

      — Otherwise, it is unspecified whether it is **ASSIGNMENT_WORD** or **WORD** that is returned.

   Assignment to the **NAME** shall occur as specified in Section 2.9.1 (on page 2248).

8. [**NAME** in function]

   When the **TOKEN** is exactly a reserved word, the token identifier for that reserved word shall result. Otherwise, when the **TOKEN** meets the requirements for a name, the token identifier **NAME** shall result. Otherwise, rule 7 applies.

9. [Body of function]

   Word expansion and assignment shall never occur, even when required by the rules above, when this rule is being parsed. Each **TOKEN** that might either be expanded or have assignment applied to it shall instead be returned as a single **WORD** consisting only of characters that are exactly the token described in Section 2.3 (on page 2233).

```
/* -------------------------------------------------------
   The grammar symbols
   ------------------------------------------------------- */
%token  WORD
%token  ASSIGNMENT_WORD
%token  NAME
%token  NEWLINE
%token  IO_NUMBER
/* The following are the operators mentioned above. */
%token  AND_IF    OR_IF    DSEMI
/*      '&&'      '||'     ';;'    */
%token  DLESS  DGREAT  LESSAND  GREATAND  LESSGREAT  DLESSDASH
/*      '<<'   '>>'    '<&'     '>&'      '<>'       '<<-'   */
%token  CLOBBER
/*      '>|'   */
/* The following are the reserved words. */
%token  If    Then    Else    Elif    Fi    Do    Done
/*      'if'  'then'  'else'  'elif'  'fi'  'do'  'done'   */
%token  Case    Esac    While    Until    For
/*      'case'  'esac'  'while'  'until'  'for'   */
/* These are reserved words, not operator tokens, and are
   recognized when reserved words are recognized. */
%token  Lbrace    Rbrace    Bang
/*      '{'       '}'       '!'   */
%token  In
/*      'in'   */
/* -------------------------------------------------------
   The Grammar
   ------------------------------------------------------- */
%start  complete_command
%%
complete_command : list separator
                 | list
                 ;
list             : list separator_op and_or
                 |                   and_or
                 ;
and_or           :                         pipeline
                 | and_or AND_IF linebreak pipeline
                 | and_or OR_IF  linebreak pipeline
                 ;
pipeline         :      pipe_sequence
                 | Bang pipe_sequence
                 ;
pipe_sequence    :                             command
                 | pipe_sequence '|' linebreak command
```

```
                 ;
command          : simple_command
                 | compound_command
                 | compound_command redirect_list
                 | function_definition
                 ;
compound_command : brace_group
                 | subshell
                 | for_clause
                 | case_clause
                 | if_clause
                 | while_clause
                 | until_clause
                 ;
subshell         : ’(’ compound_list ’)’
                 ;
compound_list    :              term
                 | newline_list term
                 |              term separator
                 | newline_list term separator
                 ;
term             : term separator and_or
                 |                and_or
                 ;
for_clause       : For name linebreak                            do_group
                 | For name linebreak in          sequential_sep_do_group
                 | For name linebreak in wordlist sequential_sep do_group
                 ;
name             : NAME                     /* Apply rule 5 */
                 ;
in               : In                       /* Apply rule 6 */
                 ;
wordlist         : wordlist WORD
                 |          WORD
                 ;
case_clause      : Case WORD linebreak in linebreak case_list Esac
                 | Case WORD linebreak in linebreak case_list_ns Esac
                 | Case WORD linebreak in linebreak Esac
                 ;
case_list_ns     : case_list case_item_ns
                 | case_item_ns
                 ;
case_list        : case_list case_item
                 | case_item
                 ;
case_item_ns     : pattern ’)’ linebreak linebreak
                 | pattern ’)’ compound_list linebreak
                 | ’(’ pattern ’)’ linebreak linebreak
                 | ’(’ pattern ’)’ compound_list linebreak
                 ;
case_item        : pattern ’)’ linebreak DSEMI linebreak
                 | pattern ’)’ compound_list linebreak
```

```
                 | ’(’ pattern ’)’ linebreak linebreak
                 | ’(’ pattern ’)’ compound_list linebreak
                 ;
pattern          :             WORD         /* Apply rule 4 */
                 | pattern ’|’ WORD         /* Do not apply rule (4) */
                 ;
if_clause        : If compound_list Then compound_list else_part Fi
                 | If compound_list Then compound_list           Fi
                 ;
else_part        : Elif compound_list Then else_part
                 | Else compound_list
                 ;
while_clause     : While compound_list do_group
                 ;
until_clause     : Until compound_list do_group
                 ;
function_definition : fname ’(’ ’)’ linebreak function_body
                 ;
function_body    : compound_command                /* Apply rule 9 */
                 | compound_command redirect_list  /* Apply rule 9 */
                 ;
fname            : NAME                            /* Apply rule 8 */
                 ;
brace_group      : Lbrace compound_list Rbrace
                 ;
do_group         : Do compound_list Done
                 ;
simple_command   : cmd_prefix cmd_word cmd_suffix
                 | cmd_prefix cmd_word
                 | cmd_prefix
                 | cmd_name cmd_suffix
                 | cmd_name
                 ;
cmd_name         : WORD                   /* Apply rule 7a */
                 ;
cmd_word         : WORD                   /* Apply rule 7b */
                 ;
cmd_prefix       :            io_redirect
                 | cmd_prefix io_redirect
                 |            ASSIGNMENT_WORD
                 | cmd_prefix ASSIGNMENT_WORD
                 ;
cmd_suffix       :            io_redirect
                 | cmd_suffix io_redirect
                 |            WORD
                 | cmd_suffix WORD
                 ;
redirect_list    :               io_redirect
                 | redirect_list io_redirect
                 ;
io_redirect      :           io_file
                 | IO_NUMBER io_file
```

```
                    |           io_here
                    | IO_NUMBER io_here
                    ;
io_file             : '<'       filename
                    | LESSAND   filename
                    | '>'       filename
                    | GREATAND  filename
                    | DGREAT    filename
                    | LESSGREAT filename
                    | CLOBBER   filename
                    ;
filename            : WORD                      /* Apply rule 2 */
                    ;
io_here             : DLESS     here_end
                    | DLESSDASH here_end
                    ;
here_end            : WORD                      /* Apply rule 3 */
                    ;
newline_list        :              NEWLINE
                    | newline_list NEWLINE
                    ;
linebreak           : newline_list
                    | /* empty */
                    ;
separator_op        : '&'
                    | ';'
                    ;
separator           : separator_op linebreak
                    | newline_list
                    ;
sequential_sep      : ';' linebreak
                    | newline_list
                    ;
```

2.11 Signals and Error Handling

When a command is in an asynchronous list, the shell shall prevent SIGQUIT and SIGINT signals from the keyboard from interrupting the command. Otherwise, signals shall have the values inherited by the shell from its parent (see also the *trap* (on page 2297) special built-in).

When a signal for which a trap has been set is received while the shell is waiting for the completion of a utility executing a foreground command, the trap associated with that signal shall not be executed until after the foreground command has completed. When the shell is waiting, by means of the *wait* utility, for asynchronous commands to complete, the reception of a signal for which a trap has been set shall cause the *wait* utility to return immediately with an exit status >128, immediately after which the trap associated with that signal shall be taken.

If multiple signals are pending for the shell for which there are associated trap actions, the order of execution of trap actions is unspecified.

2.12 Shell Execution Environment

A shell execution environment consists of the following:

- Open files inherited upon invocation of the shell, plus open files controlled by *exec*
- Working directory as set by *cd*
- File creation mask set by *umask*
- Current traps set by *trap*
- Shell parameters that are set by variable assignment (see the *set* (on page 2287) special built-in) or from the System Interfaces volume of IEEE Std 1003.1-200x environment inherited by the shell when it begins (see the *export* (on page 2281) special built-in)
- Shell functions; see Section 2.9.5 (on page 2256)
- Options turned on at invocation or by *set*
- Process IDs of the last commands in asynchronous lists known to this shell environment; see Section 2.9.3.1 (on page 2252)
- Shell aliases; see Section 2.3.1 (on page 2234)

Utilities other than the special built-ins (see Section 2.14 (on page 2266)) shall be invoked in a separate environment that consists of the following. The initial value of these objects shall be the same as that for the parent shell, except as noted below.

- Open files inherited on invocation of the shell, open files controlled by the *exec* special built-in plus any modifications, and additions specified by any redirections to the utility
- Current working directory
- File creation mask
- If the utility is a shell script, traps caught by the shell shall be set to the default values and traps ignored by the shell shall be set to be ignored by the utility; if the utility is not a shell script, the trap actions (default or ignore) shall be mapped into the appropriate signal handling actions for the utility
- Variables with the *export* attribute, along with those explicitly exported for the duration of the command, shall be passed to the utility as System Interfaces volume of IEEE Std 1003.1-200x environment variables

The environment of the shell process shall not be changed by the utility unless explicitly specified by the utility description (for example, *cd* and *umask*).

A subshell environment shall be created as a duplicate of the shell environment, except that signal traps set by that shell environment shall be set to the default values. Changes made to the subshell environment shall not affect the shell environment. Command substitution, commands that are grouped with parentheses, and asynchronous lists shall be executed in a subshell environment. Additionally, each command of a multi-command pipeline is in a subshell environment; as an extension, however, any or all commands in a pipeline may be executed in the current environment. All other commands shall be executed in the current shell environment.

2.13 Pattern Matching Notation

The pattern matching notation described in this section is used to specify patterns for matching strings in the shell. Historically, pattern matching notation is related to, but slightly different from, the regular expression notation described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions. For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation, modified to include backslash escape processing.

2.13.1 Patterns Matching a Single Character

The following *patterns matching a single character* shall match a single character: *ordinary characters*, *special pattern characters*, and *pattern bracket expressions*. The pattern bracket expression also shall match a single collating element. A backslash character shall escape the following character. The escaping backslash shall be discarded.

An ordinary character is a pattern that shall match itself. It can be any character in the supported character set except for NUL, those special shell characters in Section 2.2 (on page 2232) that require quoting, and the following three special pattern characters. Matching shall be based on the bit pattern used for encoding the character, not on the graphic representation of the character. If any character (ordinary, shell special, or pattern special) is quoted, that pattern shall match the character itself. The shell special characters always require quoting.

When unquoted and outside a bracket expression, the following three characters shall have special meaning in the specification of patterns:

? A question-mark is a pattern that shall match any character.

* An asterisk is a pattern that shall match multiple characters, as described in Section 2.13.2.

[The open bracket shall introduce a pattern bracket expression.

The description of basic regular expression bracket expressions in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.5, RE Bracket Expression shall also apply to the pattern bracket expression, except that the exclamation mark character ('!') shall replace the circumflex character ('^') in its role in a *non-matching list* in the regular expression notation. A bracket expression starting with an unquoted circumflex character produces unspecified results.

When pattern matching is used where shell quote removal is not performed (such as in the argument to the *find name* primary when *find* is being called using one of the *exec* functions as defined in the System Interfaces volume of IEEE Std 1003.1-200x, or in the *pattern* argument to the *fnmatch*( ) function), special characters can be escaped to remove their special meaning by preceding them with a backslash character. This escaping backslash is discarded. The sequence "\\" represents one literal backslash. All of the requirements and effects of quoting on ordinary, shell special, and special pattern characters shall apply to escaping in this context.

2.13.2 Patterns Matching Multiple Characters

The following rules are used to construct *patterns matching multiple characters* from *patterns matching a single character*:

1. The asterisk ('*') is a pattern that shall match any string, including the null string.
2. The concatenation of *patterns matching a single character* is a valid pattern that shall match the concatenation of the single characters or collating elements matched by each of the concatenated patterns.

3. The concatenation of one or more *patterns matching a single character* with one or more asterisks is a valid pattern. In such patterns, each asterisk shall match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

2.13.3 Patterns Used for Filename Expansion

The rules described so far in Section 2.13.1 (on page 2264) and Section 2.13.2 (on page 2264) are qualified by the following rules that apply when pattern matching notation is used for filename expansion:

1. The slash character in a pathname shall be explicitly matched by using one or more slashes in the pattern; it shall neither be matched by the asterisk or question-mark special characters nor by a bracket expression. Slashes in the pattern shall be identified before bracket expressions; thus, a slash cannot be included in a pattern bracket expression used for filename expansion. If a slash character is found following an unescaped open square bracket character before a corresponding closing square bracket is found, the open bracket shall be treated as an ordinary character. For example, the pattern `"a[b/c]d"` does not match such pathnames as **abd** or **a/d**. It only matches a pathname of literally **a[b/c]d**.

2. If a filename begins with a period (`'.'`), the period shall be explicitly matched by using a period as the first character of the pattern or immediately following a slash character. The leading period shall not be matched by:

   - The asterisk or question-mark special characters
   - A bracket expression containing a non-matching list, such as `"[!a]"`, a range expression, such as `"[%−0]"`, or a character class expression, such as `"[[:punct:]]"`

   It is unspecified whether an explicit period in a bracket expression matching list, such as `"[.abc]"`, can match a leading period in a filename.

3. Specified patterns shall be matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character shall require read permission in the directory containing that component. Any component, except the last, that does not contain a pattern character shall require search permission. For example, given the pattern:

   ```
   /foo/bar/x*/bam
   ```

   search permission is needed for directories **/** and **foo**, search and read permissions are needed for directory **bar**, and search permission is needed for each **x*** directory. If the pattern matches any existing filenames or pathnames, the pattern shall be replaced with those filenames and pathnames, sorted according to the collating sequence in effect in the current locale. If the pattern contains an invalid bracket expression or does not match any existing filenames or pathnames, the pattern string shall be left unchanged.

2.14 Special Built-In Utilities

The following *special built-in* utilities shall be supported in the shell command language. The output of each command, if any, shall be written to standard output, subject to the normal redirection and piping possible with all commands.

The term *built-in* implies that the shell can execute the utility directly and does not need to search for it. An implementation may choose to make any utility a built-in; however, the special built-in utilities described here differ from regular built-in utilities in two respects:

1. A syntax error in a special built-in utility may cause a shell executing that utility to abort, while a syntax error in a regular built-in utility shall not cause a shell executing that utility to abort. (See Section 2.8.1 (on page 2247) for the consequences of errors on interactive and non-interactive shells.) If a special built-in utility encountering a syntax error does not abort the shell, its exit value shall be non-zero.
2. Variable assignments specified with special built-in utilities remain in effect after the built-in completes; this shall not be the case with a regular built-in or other utility.

The special built-in utilities in this section need not be provided in a manner accessible via the *exec* family of functions defined in the System Interfaces volume of IEEE Std 1003.1-200x.

Some of the special built-ins are described as conforming to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines. For those that are not, the requirement in Section 1.11 (on page 2221) that "−−" be recognized as a first argument to be discarded does not apply and a conforming application shall not use that argument.

NAME

break — exit from for, while, or until loop

SYNOPSIS

```
break [n]
```

DESCRIPTION

The *break* utility shall exit from the smallest enclosing **for**, **while**, or **until** loop, if any; or from the *n*th enclosing loop if *n* is specified. The value of *n* is an unsigned decimal integer greater than or equal to 1. The default shall be equivalent to *n*=1. If *n* is greater than the number of enclosing loops, the outermost enclosing loop shall be exited. Execution shall continue with the command immediately following the loop.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

0	Successful completion.
>0	The *n* value was not an unsigned decimal integer greater than or equal to 1.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
for i in * do
    if test -d "$i" then break fi done
```

RATIONALE

In early proposals, consideration was given to expanding the syntax of *break* and *continue* to refer to a label associated with the appropriate loop as a preferable alternative to the *n* method. However, this volume of IEEE Std 1003.1-200x does reserve the namespace of command names ending with a colon. It is anticipated that a future implementation could take advantage of this and provide something like:

```
outofloop: for i in a b c d e
do
    for j in 0 1 2 3 4 5 6 7 8 9
    do
        if test -r "${i}${j}"
        then break outofloop
        fi
    done
done
```

and that this might be standardized after implementation experience is achieved.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

colon — null utility

SYNOPSIS

`: [argument ...]`

DESCRIPTION

This utility shall only expand command *arguments*. It is used when a command is needed, as in the *then* condition of an **if** command, but nothing is to be done by the command.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Zero.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
: ${X=abc}
if     false
then   :
else   echo $X
fi
abc
```

As with any of the special built-ins, the null utility can also have variable assignments and redirections associated with it, such as:

```
x=y : > z
```

which sets variable *x* to the value *y* (so that it persists after the null utility completes) and creates or truncates file **z**.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

continue — continue for, while, or until loop

SYNOPSIS

`continue [n]`

DESCRIPTION

The *continue* utility shall return to the top of the smallest enclosing **for**, **while**, or **until** loop, or to the top of the *n*th enclosing loop, if *n* is specified. This involves repeating the condition list of a **while** or **until** loop or performing the next assignment of a **for** loop, and reexecuting the loop if appropriate.

The value of *n* is a decimal integer greater than or equal to 1. The default shall be equivalent to *n*=1. If *n* is greater than the number of enclosing loops, the outermost enclosing loop shall be used.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

0	Successful completion.
>0	The *n* value was not an unsigned decimal integer greater than or equal to 1.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
for i in *
do
    if test -d "$i"
    then continue
    fi
    echo "\"$i\"" is not a directory.
done
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

dot — execute commands in current environment

SYNOPSIS

```
. file
```

DESCRIPTION

The shell shall execute commands from the *file* in the current environment.

If *file* does not contain a slash, the shell shall use the search path specified by *PATH* to find the directory containing *file*. Unlike normal command search, however, the file searched for by the *dot* utility need not be executable. If no readable file is found, a non-interactive shell shall abort; an interactive shell shall write a diagnostic message to standard error, but this condition shall not be considered a syntax error.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Returns the value of the last command executed, or a zero exit status if no command is executed.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
cat foobar
foo=hello bar=world
. foobar
echo $foo $bar
hello world
```

RATIONALE

Some older implementations searched the current directory for the *file*, even if the value of *PATH* disallowed it. This behavior was omitted from this volume of IEEE Std 1003.1-200x due to concerns about introducing the susceptibility to trojan horses that the user might be trying to avoid by leaving *dot* out of *PATH*.

The KornShell version of *dot* takes optional arguments that are set to the positional parameters. This is a valid extension that allows a *dot* script to behave identically to a function.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

eval — construct command by concatenating arguments

SYNOPSIS

```
eval [argument ...]
```

DESCRIPTION

The *eval* utility shall construct a command by concatenating *argument*s together, separating each with a <space>. The constructed command shall be read and executed by the shell.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If there are no *argument*s, or only null arguments, *eval* shall return a zero exit status; otherwise, it shall return the exit status of the command defined by the string of concatenated *argument*s separated by spaces.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
foo=10 x=foo
y='$'$x
echo $y
$foo
eval y='$'$x
echo $y
10
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

exec — execute commands and open, close, or copy file descriptors

SYNOPSIS

`exec [command [argument ...]]`

DESCRIPTION

The *exec* utility shall open, close, and/or copy file descriptors as specified by any redirections as part of the command.

If *exec* is specified without *command* or *argument*s, and any file descriptors with numbers greater than 2 are opened with associated redirection statements, it is unspecified whether those file descriptors remain open when the shell invokes another utility. Scripts concerned that child shells could misuse open file descriptors can always close them explicitly, as shown in one of the following examples.

If *exec* is specified with *command*, it shall replace the shell with *command* without creating a new process. If *argument*s are specified, they shall be arguments to *command*. Redirection affects the current shell execution environment.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If *command* is specified, *exec* shall not return to the shell; rather, the exit status of the process shall be the exit status of the program implementing *command*, which overlaid the shell. If *command* is not found, the exit status shall be 127. If *command* is found, but it is not an executable utility, the exit status shall be 126. If a redirection error occurs (see Section 2.8.1 (on page 2247)), the shell shall exit with a value in the range 1–125. Otherwise, *exec* shall return a zero exit status.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

Open *readfile* as file descriptor 3 for reading:

```
exec 3< readfile
```

Open *writefile* as file descriptor 4 for writing:

```
exec 4> writefile
```

Make file descriptor 5 a copy of file descriptor 0:

```
exec 5<&0
```

Close file descriptor 3:

```
exec 3<&–
```

Cat the file **maggie** by replacing the current shell with the *cat* utility:

```
exec cat maggie
```

RATIONALE

Most historical implementations were not conformant in that:

```
foo=bar exec cmd
```

did not pass **foo** to **cmd**.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

exit — cause the shell to exit

SYNOPSIS

`exit [n]`

DESCRIPTION

The *exit* utility shall cause the shell to exit with the exit status specified by the unsigned decimal integer *n*. If *n* is specified, but its value is not between 0 and 255 inclusively, the exit status is undefined.

A *trap* on **EXIT** shall be executed before the shell terminates, except when the *exit* utility is invoked in that *trap* itself, in which case the shell shall exit immediately.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The exit status shall be *n*, if specified. Otherwise, the value shall be the exit value of the last command executed, or zero if no command was executed. When *exit* is executed in a *trap* action, the last command is considered to be the command that executed immediately preceding the *trap* action.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

Exit with a *true* value:

```
exit 0
```

Exit with a *false* value:

```
exit 1
```

RATIONALE

As explained in other sections, certain exit status values have been reserved for special uses and should be used by applications only for those purposes:

126	A file to be executed was found, but it was not an executable utility.
127	A utility to be executed was not found.
>128	A command was interrupted by a signal.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

export — set export attribute for variables

SYNOPSIS

```
export name[=word]...
```

```
export −p
```

DESCRIPTION

The shell shall give the export attribute to the variables corresponding to the specified *name*s, which shall cause them to be in the environment of subsequently executed commands.

The *export* special built-in shall support the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

When **−p** is specified, *export* shall write to the standard output the names and values of all exported variables, in the following format:

```
"export %s=%s\n", <name>, <value>
```

if *name* is set, and:

```
"export %s\n", <name>
```

if *name* is unset.

The shell shall format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same exporting results, except:

1. Read-only variables with values cannot be reset.
2. Variables that were unset at the time they were output need not be reset to the unset state if a value is assigned to the variable between the time the state was saved and the time at which the saved output is reinput to the shell.

When no arguments are given, the results are unspecified.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Zero.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

Export *PWD* and *HOME* variables:

```
export PWD HOME
```

Set and export the *PATH* variable:

```
export PATH=/local/bin:$PATH
```

Save and restore all exported variables:

```
export −p > temp-file
unset a lot of variables
... processing
. temp-file
```

RATIONALE

Some historical shells use the no-argument case as the functional equivalent of what is required here with **−p**. This feature was left unspecified because it is not historical practice in all shells, and some scripts may rely on the now-unspecified results on their implementations. Attempts to specify the **−p** output as the default case were unsuccessful in achieving consensus. The **−p** option was added to allow portable access to the values that can be saved and then later restored using; for example, a *dot* script.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

Issue 6

IEEE PASC Interpretation 1003.2 #203 is applied, clarifying the format when a variable is unset.

NAME

readonly — set read-only attribute for variables

SYNOPSIS

```
readonly name[=word]...
```

```
readonly −p
```

DESCRIPTION

The variables whose *name*s are specified shall be given the *readonly* attribute. The values of variables with the *readonly* attribute cannot be changed by subsequent assignment, nor can those variables be unset by the *unset* utility.

The *readonly* special built-in shall support the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

When **−p** is specified, *readonly* writes to the standard output the names and values of all read-only variables, in the following format:

```
"readonly %s=%s\n", <name>, <value>
```

if *name* is set, and |

```
"readonly %s\n", <name>
```

|

if *name* is unset. |

The shell shall format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same value and read-only attribute-setting results in a shell execution environment in which: |

1. Variables with values at the time they were output do not have the read-only attribute set. |
2. Variables that were unset at the time they were output do not have a value at the time at which the saved output is reinput to the shell. |

When no arguments are given, the results are unspecified. |

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Zero.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
readonly HOME PWD
```

RATIONALE

Some historical shells preserve the read-only attribute across separate invocations. This volume of IEEE Std 1003.1-200x allows this behavior, but does not require it.

The −**p** option allows portable access to the values that can be saved and then later restored using; for example, a *dot* script. Also see the RATIONALE for *export* (on page 2281) for a description of the no-argument and −**p** output cases and a related example.

Read-only functions were considered, but they were omitted as not being historical practice or particularly useful. Furthermore, functions must not be *readonly* across invocations to preclude *spoofing* (spoofing is the term for the practice of creating a program that acts like a well-known utility with the intent of subverting the real intent of the user) of administrative or security-relevant (or security-conscious) shell scripts.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

Issue 6

IEEE PASC Interpretation 1003.2 #203 is applied, clarifying the format when a variable is unset.

NAME

return — return from a function

SYNOPSIS

```
return [n]
```

DESCRIPTION

The *return* utility shall cause the shell to stop executing the current function or *dot* script. If the shell is not currently executing a function or *dot* script, the results are unspecified.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The value of the special parameter `'?'` shall be set to *n*, an unsigned decimal integer, or to the exit status of the last command executed if *n* is not specified. If the value of *n* is greater than 255, the results are undefined. When *return* is executed in a *trap* action, the last command is considered to be the command that executed immediately preceding the *trap* action.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The behavior of *return* when not in a function or *dot* script differs between the System V shell and the KornShell. In the System V shell this is an error, whereas in the KornShell, the effect is the same as *exit*.

The results of returning a number greater than 255 are undefined because of differing practices in the various historical implementations. Some shells AND out all but the low-order 8 bits; others allow larger values, but not of unlimited size.

See the discussion of appropriate exit status values under *exit* (on page 2279).

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

set — set or unset options and positional parameters

SYNOPSIS

XSI
```
set [-abCefmnuvx][-h][-o option][argument...]
set [+abCefmnuvx][+h][+o option][argument...]
set --[argument...]
set -o
set +o
```

DESCRIPTION

If no options or *argument*s are specified, *set* shall write the names and values of all shell variables in the collation sequence of the current locale. Each *name* shall start on a separate line, using the format:

`"%s=%s\n", <name>, <value>`

The *value* string shall be written with appropriate quoting so that it is suitable for reinput to the shell, setting or resetting, as far as possible, the variables that are currently set. Read-only variables cannot be reset; see the description of shell quoting in Section 2.2 (on page 2232).

When options are specified, they shall set or unset attributes of the shell, as described below. When *argument*s are specified, they cause positional parameters to be set or unset, as described below. Setting or unsetting attributes and positional parameters are not necessarily related actions, but they can be combined in a single invocation of *set*.

The *set* special built-in shall support the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines except that options can be specified with either a leading hyphen (meaning enable the option) or plus sign (meaning disable it).

Implementations shall support the options in the following list in both their hyphen and plus-sign forms. These options can also be specified as options to *sh*.

–a When this option is on, the export attribute shall be set for each variable to which an assignment is performed; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.21, Variable Assignment. If the assignment precedes a utility name in a command, the export attribute shall not persist in the current execution environment after the utility completes, with the exception that preceding one of the special built-in utilities causes the export attribute to persist after the built-in has completed. If the assignment does not precede a utility name in the command, or if the assignment is a result of the operation of the *getopts* or *read* utilities, the export attribute shall persist until the variable is unset.

–b This option shall be supported if the implementation supports the User Portability Utilities option. It shall cause the shell to notify the user asynchronously of background job completions. The following message is written to standard error:

`"[%d]%c %s%s\n", <job-number>, <current>, <status>, <job-name>`

where the fields shall be as follows:

<current> The character '+' identifies the job that would be used as a default for the *fg* or *bg* utilities; this job can also be specified using the *job_id* "%+" or "%%". The character '–' identifies the job that would become the default if the current default job were to exit; this job can also be specified using the *job_id* "%–". For other jobs, this field is a <space>. At most one job can be identified with '+' and at most one job can be identified with '–'.

If there is any suspended job, then the current job shall be a suspended job. If there are at least two suspended jobs, then the previous job also shall be a suspended job.

<job-number> A number that can be used to identify the process group to the *wait*, *fg*, *bg*, and *kill* utilities. Using these utilities, the job can be identified by prefixing the job number with `'%'`.

<status> Unspecified.

<job-name> Unspecified.

When the shell notifies the user a job has been completed, it may remove the job's process ID from the list of those known in the current shell execution environment; see Section 2.9.3.1 (on page 2252). Asynchronous notification shall not be enabled by default.

−C (Uppercase C.) Prevent existing files from being overwritten by the shell's `'>'` redirection operator (see Section 2.7.2 (on page 2245)); the `">|"` redirection operator shall override this *noclobber* option for an individual file.

−e When this option is on, if a simple command fails for any of the reasons listed in Section 2.8.1 (on page 2247) or returns an exit status value >0, and is not part of the compound list following a **while**, **until**, or **if** keyword, and is not a part of an AND or OR list, and is not a pipeline preceded by the **!** reserved word, then the shell shall immediately exit.

−f The shell shall disable pathname expansion.

XSI **−h** Locate and remember utilities invoked by functions as those functions are defined (the utilities are normally located when the function is executed).

−m This option shall be supported if the implementation supports the User Portability Utilities option. All jobs shall be run in their own process groups. Immediately before the shell issues a prompt after completion of the background job, a message reporting the exit status of the background job shall be written to standard error. If a foreground job stops, the shell shall write a message to standard error to that effect, formatted as described by the *jobs* utility. In addition, if a job changes status other than exiting (for example, if it stops for input or output or is stopped by a SIGSTOP signal), the shell shall write a similar message immediately prior to writing the next prompt. This option is enabled by default for interactive shells.

−n The shell shall read commands but does not execute them; this can be used to check for shell script syntax errors. An interactive shell may ignore this option.

−o Write the current settings of the options to standard output in an unspecified format.

+o Write the current option settings to standard output in a format that is suitable for reinput to the shell as commands that achieve the same options settings.

−o *option*

This option is supported if the system supports the User Portability Utilities option. It shall set various options, many of which shall be equivalent to the single option letters. The following values of *option* shall be supported:

allexport Equivalent to **−a**.

errexit Equivalent to **−e**.

ignoreeof Prevent an interactive shell from exiting on end-of-file. This setting prevents accidental logouts when <control>-D is entered. A user shall explicitly *exit* to leave the interactive shell.

monitor	Equivalent to **–m**. This option is supported if the system supports the User Portability Utilities option.
noclobber	Equivalent to **–C** (uppercase C).
noglob	Equivalent to **–f**.
noexec	Equivalent to **–n**.
nolog	Prevent the entry of function definitions into the command history; see **Command History List** (on page 3052).
notify	Equivalent to **–b**.
nounset	Equivalent to **–u**.
verbose	Equivalent to **–v**.
vi	Allow shell command line editing using the built-in *vi* editor. Enabling *vi* mode shall disable any other command line editing mode provided as an implementation extension. It need not be possible to set *vi* mode on for certain block-mode terminals.
xtrace	Equivalent to **–x**.

–u The shell shall write a message to standard error when it tries to expand a variable that is not set and immediately exit. An interactive shell shall not exit.

–v The shell shall write its input to standard error as it is read.

–x The shell shall write to standard error a trace for each command after it expands the command and before it executes it. It is unspecified whether the command that turns tracing off is traced.

The default for all these options shall be off (unset) unless the shell was invoked with them on; see *sh*.

The remaining arguments shall be assigned in order to the positional parameters. The special parameter '#' shall be set to reflect the number of positional parameters. All positional parameters shall be unset before any new values are assigned.

The special argument "––" immediately following the *set* command name can be used to delimit the arguments if the first argument begins with '+' or '–', or to prevent inadvertent listing of all shell variables when there are no arguments. The command *set*–– without *argument* shall unset all positional parameters and set the special parameter '#' to zero.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Zero.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

Write out all variables and their values:

```
set
```

Set $1, $2, and $3 and set "$#" to 3:

```
set c a b
```

Turn on the –**x** and –**v** options:

```
set –xv
```

Unset all positional parameters:

```
set ––
```

Set $1 to the value of –**x**, even if **x** begins with '–' or '+':

```
set –– "$x"
```

Set the positional parameters to the expansion of **x**, even if **x** expands with a leading '–' or '+':

```
set –– $x
```

RATIONALE

The *set* –– form is listed specifically in the SYNOPSIS even though this usage is implied by the Utility Syntax Guidelines. The explanation of this feature removes any ambiguity about whether the *set* –– form might be misinterpreted as being equivalent to *set* without any options or arguments. The functionality of this form has been adopted from the KornShell. In System V, *set* –– only unsets parameters if there is at least one argument; the only way to unset all parameters is to use *shift*. Using the KornShell version should not affect System V scripts because there should be no reason to issue it without arguments deliberately; if it were issued as, for example:

```
set –– "$@"
```

and there were in fact no arguments resulting from `"$@"`, unsetting the parameters would have no result.

The *set* + form in early proposals was omitted as being an unnecessary duplication of *set* alone and not widespread historical practice.

The *noclobber* option was changed to allow *set* −**C** as well as the *set* −**o** *noclobber* option. The single-letter version was added so that the historical `"$−"` paradigm would not be broken; see Section 2.5.2 (on page 2235).

The −**h** flag is related to command name hashing and is only required on XSI-conformant systems.

The following *set* flags were omitted intentionally with the following rationale:

−**k** The −**k** flag was originally added by the author of the Bourne shell to make it easier for users of pre-release versions of the shell. In early versions of the Bourne shell the construct *set name*=*value*, had to be used to assign values to shell variables. The problem with −**k** is that the behavior affects parsing, virtually precluding writing any compilers. To explain the behavior of −**k**, it is necessary to describe the parsing algorithm, which is implementation-defined. For example:

```
set −k; echo name=value
```

and:

```
set x−−k
echo name=value
```

behave differently. The interaction with functions is even more complex. What is more, the −**k** flag is never needed, since the command line could have been reordered.

−**t** The −**t** flag is hard to specify and almost never used. The only known use could be done with here-documents. Moreover, the behavior with *ksh* and *sh* differs. The reference page says that it exits after reading and executing one command. What is one command? If the input is *date*;*date*, *sh* executes both *date* commands while *ksh* does only the first.

Consideration was given to rewriting *set* to simplify its confusing syntax. A specific suggestion was that the *unset* utility should be used to unset options instead of using the non-*getopt*( )-able +*option* syntax. However, the conclusion was reached that the historical practice of using +*option* was satisfactory and that there was no compelling reason to modify such widespread historical practice.

The −**o** option was adopted from the KornShell to address user needs. In addition to its generally friendly interface, −**o** is needed to provide the *vi* command line editing mode, for which historical practice yields no single-letter option name. (Although it might have been possible to invent such a letter, it was recognized that other editing modes would be developed and −**o** provides ample name space for describing such extensions.)

Historical implementations are inconsistent in the format used for −**o** option status reporting. The **+o** format without an option-argument was added to allow portable access to the options that can be saved and then later restored using, for instance, a dot script.

Historically, *sh* did trace the command *set* **+x**, but *ksh* did not.

The *ignoreeof* setting prevents accidental logouts when the end-of-file character (typically <control>-D) is entered. A user shall explicitly *exit* to leave the interactive shell.

The *set* −**m** option was added to apply only to the UPE because it applies primarily to interactive use, not shell script applications.

The ability to do asynchronous notification became available in the 1988 version of the KornShell. To have it occur, the user had to issue the command:

```
trap "jobs –n" CLD
```

The C shell provides two different levels of an asynchronous notification capability. The environment variable *notify* is analogous to what is done in *set* –**b** or *set* –**o** *notify*. When set, it notifies the user immediately of background job completions. When unset, this capability is turned off.

The other notification ability comes through the built-in utility *notify*. The syntax is:

```
notify [%job ... ]
```

By issuing *notify* with no operands, it causes the C shell to notify the user asynchronously when the state of the current job changes. If given operands, *notify* asynchronously informs the user of changes in the states of the specified jobs.

To add asynchronous notification to the POSIX shell, neither the KornShell extensions to *trap*, nor the C shell *notify* environment variable seemed appropriate (*notify* is not a proper POSIX environment variable name).

The *set* –**b** option was selected as a compromise.

The *notify* built-in was considered to have more functionality than was required for simple asynchronous notification.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

Issue 6

The obsolescent *set* command name followed by `'–'` has been removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *nolog* option is added to *set* –**o**.

NAME

shift — shift positional parameters

SYNOPSIS

```
shift [n]
```

DESCRIPTION

The positional parameters shall be shifted. Positional parameter 1 shall be assigned the value of parameter (1+*n*), parameter 2 shall be assigned the value of parameter (2+*n*), and so on. The parameters represented by the numbers `"$#"` down to `"$#−n+1"` shall be unset, and the parameter `'#'` is updated to reflect the new number of positional parameters.

The value *n* shall be an unsigned decimal integer less than or equal to the value of the special parameter `'#'`. If *n* is not given, it shall be assumed to be 1. If *n* is 0, the positional and special parameters are not changed.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The exit status is >0 if *n*>$#; otherwise, it is zero.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
$ set a b c d e
$ shift 2
$ echo $*
c d e
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

times — write process times

SYNOPSIS

`times`

DESCRIPTION

Write the accumulated user and system times for the shell and for all of its child processes, in the following POSIX locale format:

```
"%dm%fs %dm%fs\n%dm%fs %dm%fs\n", <shell user minutes>,
    <shell user seconds>, <shell system minutes>,
    <shell system seconds>, <children user minutes>,
    <children user seconds>, <children system minutes>,
    <children system seconds>
```

The four pairs of times shall correspond to the members of the **<sys/times.h>** **tms** structure (defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers) as returned by *times*( ): *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime*, respectively. |

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Zero.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

```
$ times
0m0.43s 0m1.11s
8m44.18s 1m43.23s
```

RATIONALE

The *times* special built-in from the Single UNIX Specification is now required for all conforming shells.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

NAME

trap — trap signals

SYNOPSIS

```
trap [action condition ...]
```

DESCRIPTION

If *action* is '−', the shell shall reset each *condition* to the default value. If *action* is null (""), the shell shall ignore each specified *condition* if it arises. Otherwise, the argument *action* shall be read and executed by the shell when one of the corresponding conditions arises. The action of *trap* shall override a previous action (either default action or one explicitly set). The value of "$?" after the *trap* action completes shall be the value it had before *trap* was invoked.

The condition can be EXIT, 0 (equivalent to EXIT), or a signal specified using a symbolic name, without the SIG prefix, as listed in the tables of signal names in the **<signal.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers; for example, HUP, INT, QUIT, TERM. Implementations may permit lowercase signal names or names with the SIG prefix as an extension. Setting a trap for SIGKILL or SIGSTOP produces undefined results.

The environment in which the shell executes a *trap* on EXIT shall be identical to the environment immediately after the last command executed before the *trap* on EXIT was taken.

Each time *trap* is invoked, the *action* argument shall be processed in a manner equivalent to:

```
eval "$action"
```

Signals that were ignored on entry to a non-interactive shell cannot be trapped or reset, although no error need be reported when attempting to do so. An interactive shell may reset or catch signals ignored on entry. Traps shall remain in place for a given shell until explicitly changed with another *trap* command.

When a subshell is entered, traps that are not being ignored are set to the default actions. This does not imply that the *trap* command cannot be used within the subshell to set new traps.

The *trap* command with no arguments shall write to standard output a list of commands associated with each condition. The format shall be:

```
"trap -- %s %s ...\n", <action>, <condition> ...
```

The shell shall format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same trapping results. For example:

```
save_traps=$(trap)
...
eval "$save_traps"
```

XSI XSI-conformant systems also allow numeric signal numbers for the conditions corresponding to the following signal names:

	Signal Number	Signal Name
XSI	1	SIGHUP
XSI	2	SIGINT
XSI	3	SIGQUIT
XSI	6	SIGABRT
XSI	9	SIGKILL
XSI	14	SIGALRM
XSI	15	SIGTERM

The *trap* special built-in shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If the trap name or number is invalid, a non-zero exit status shall be returned; otherwise, zero shall be returned. For both interactive and non-interactive shells, invalid signal names or numbers shall not be considered a syntax error and do not cause the shell to abort.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

Write out a list of all traps and actions:

```
trap
```

Set a trap so the *logout* utility in the directory referred to by the *HOME* environment variable executes when the shell terminates:

```
trap '$HOME/logout' EXIT
```

or:

```
trap '$HOME/logout' 0
```

Unset traps on INT, QUIT, TERM, and EXIT:

```
trap - INT QUIT TERM EXIT
```

RATIONALE

Implementations may permit lowercase signal names as an extension. Implementations may also accept the names with the SIG prefix; no known historical shell does so. The *trap* and *kill* utilities in this volume of IEEE Std 1003.1-200x are now consistent in their omission of the SIG prefix for signal names. Some *kill* implementations do not allow the prefix, and *kill* −**l** lists the signals without prefixes.

Trapping SIGKILL or SIGSTOP is syntactically accepted by some historical implementations, but it has no effect. Portable POSIX applications cannot attempt to trap these signals.

The output format is not historical practice. Since the output of historical *trap* commands is not portable (because numeric signal values are not portable) and had to change to become so, an opportunity was taken to format the output in a way that a shell script could use to save and then later reuse a trap if it wanted.

The KornShell uses an **ERR** trap that is triggered whenever *set* −**e** would cause an exit. This is allowable as an extension, but was not mandated, as other shells have not used it.

The text about the environment for the EXIT trap invalidates the behavior of some historical versions of interactive shells which, for example, close the standard input before executing a trap on 0. For example, in some historical interactive shell sessions the following trap on 0 would always print `"--"`:

```
trap 'read foo; echo "-$foo-"' 0
```

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

Issue 6

XSI-conforming implementations provide the mapping of signal names to numbers given above (previously this had been marked obsolescent). Other implementations need not provide this optional mapping.

NAME

unset — unset values and attributes of variables and functions

SYNOPSIS

```
unset [−fv] name ...
```

DESCRIPTION

Each variable or function specified by *name* shall be unset.

If **−v** is specified, *name* refers to a variable name and the shell shall unset it and remove it from the environment. Read-only variables cannot be unset.

If **−f** is specified, *name* refers to a function and the shell shall unset the function definition.

If neither **−f** nor **−v** is specified, *name* refers to a variable; if a variable by that name does not exist, it is unspecified whether a function by that name, if any, shall be unset.

Unsetting a variable or function that was not previously set shall not be considered an error and does not cause the shell to abort.

The *unset* special built-in shall support the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

Note that:

```
VARIABLE=
```

is not equivalent to an *unset* of **VARIABLE**; in the example, **VARIABLE** is set to `""`. Also, the variables that can be *unset* should not be misinterpreted to include the special parameters (see Section 2.5.2 (on page 2235)).

OPTIONS

None.

OPERANDS

None.

STDIN

None.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

None.

STDOUT

None.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

0 All *name* operands were successfully unset.

>0 At least one *name* could not be unset.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

None.

EXAMPLES

Unset *VISUAL* variable:

```
unset −v VISUAL
```

Unset the functions **foo** and **bar**:

```
unset −f foo bar
```

RATIONALE

Consideration was given to omitting the **−f** option in favor of an *unfunction* utility, but the standard developers decided to retain historical practice.

The **−v** option was introduced because System V historically used one name space for both variables and functions. When *unset* is used without options, System V historically unset either a function or a variable, and there was no confusion about which one was intended. A portable POSIX application can use *unset* without an option to unset a variable, but not a function; the **−f** option must be used.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.14 (on page 2266)

CHANGE HISTORY

None.

Chapter 3

Batch Environment Services

BE This chapter describes the services and utilities that shall be implemented on all systems that claim conformance to the Batch Environment option. This functionality is dependent on support of this option (and the rest of this section is not further shaded for this option).

3.1 General Concepts

3.1.1 Batch Client-Server Interaction

Batch jobs are created and managed by batch servers. A batch client interacts with a batch server to access batch services on behalf of the user. In order to use batch services, a user must have access to a batch client.

A batch server is a computational entity, such as a daemon process, that provides batch services. Batch servers route, queue, modify, and execute batch jobs on behalf of batch clients.

The batch utilities described in this volume of IEEE Std 1003.1-200x (and listed in Table 3-1) are clients of batch services; they allow users to perform actions on the job such as creating, modifying, and deleting batch jobs from a shell command line. Although these batch utilities may be said to accomplish certain services, they actually obtain services on behalf of a user by means of requests to batch servers.

Table 3-1 Batch Utilities

qalter
qdel
qhold
qmove
qmsg
qrerun
qrls
qselect
qsig
qstat
qsub

Client-server interaction takes place by means of the batch requests defined in this chapter. Because direct access to batch jobs and queues is limited to batch servers, clients and servers of different implementations can interoperate, since dependencies on private structures for batch jobs and queues are limited to batch servers. Also, batch servers may be clients of other batch servers.

3.1.2 Batch Queues

Two types of batch queue are described: *routing queues* and *execution queues*. When a batch job is placed in a routing queue, it is a candidate for routing. A batch job is removed from routing queues under the following conditions:

- The batch job has been routed to another queue.
- The batch job has been deleted from the batch queue.
- The batch job has been aborted.

When a batch job is placed in an execution queue, it is a candidate for execution.

A batch job is removed from an execution queue under the following conditions:

- The batch job has been executed and exited.

- The batch job has been aborted.
- The batch job has been deleted from the batch queue.
- The batch job has been moved to another queue.

Access to a batch queue is limited to the batch server that manages the batch queue. Clients never access a batch queue or a batch job directly, either to read or write information; all client access to batch queues or jobs takes place through batch servers.

3.1.3 Batch Job Creation

When a batch server creates a batch job on behalf of a client, it shall assign a batch job identifier to the job. A batch job identifier consists of both a sequence number that is unique among the sequence numbers issued by that server and the name of the server. Since the batch server name is unique within a name space, the job identifier is likewise unique within the name space.

The batch server that creates a batch job shall return the batch server-assigned job identifier to the client that requested the job creation. If the batch server routes or moves the job to another server, it sends the job identifier with the job. Once assigned, the job identifier of a batch job shall never change.

3.1.4 Batch Job Tracking

Since a batch job may be moved after creation, the batch server name component of the job identifier need not indicate the location of the job. An implementation may provide a batch job tracking mechanism, in which case the user generally does not need to know the location of the job. However, an implementation need not provide a batch job tracking mechanism, in which case the user must find routed jobs by probing the possible destinations.

3.1.5 Batch Job Routing

To route a batch job, a batch server either moves the job to some other queue that is managed by the batch server, or requests that some other batch server accept the job.

Each routing queue has one or more queues to which it can route batch jobs. The batch server administrator creates routing queues.

A batch server may route a batch job from a routing queue to another routing queue. Batch servers shall prevent or otherwise handle cases of circular routing paths. As a deferred service, a batch server routes jobs from the routing queues that it manages. The algorithm by which a batch server selects a batch queue to which to route a batch job is implementation-defined.

A batch job need not be eligible for routing to all the batch queues fed by the routing queue from which it is routed. A batch server that has been asked to accept the job may reject the request if the job requires resources that are unavailable to that batch server, or if the client is not authorized to access the batch server.

Batch servers may route high-priority jobs before low-priority jobs, but, on other than overloaded systems, the effect may be imperceptible to the user. If all the batch servers fed by a routing queue reject requests to accept the job for reasons that are permanent, the batch server that manages the job shall abort the job. If all or some rejections are temporary, the batch server should try to route the job again at some later point.

The reasons for rejecting a batch job are implementation-defined. The reasons for which the routing should be retried later and the reasons for which the job should be aborted are also implementation-defined.

3.1.6 Batch Job Execution

To execute a batch job is to create a session leader (a process) that runs the shell program indicated by the *Shell_Path* attribute of the job. The script shall be passed to the program as its standard input. An implementation may pass the script to the program by other implementation-defined means. At the time a batch job begins execution, it is defined to enter the RUNNING state. The primary program that is executed by a batch job is typically, though not necessarily, a shell program.

A batch server shall execute eligible jobs as a deferred service—no client request is necessary once the batch job is created and eligible. However, the attributes of a batch job, such as the job hold type, may render the job ineligible. A batch server shall scan the execution queues that it manages for jobs that are eligible for execution. The algorithm by which the batch server selects eligible jobs for execution is implementation-defined.

As part of creating the process for the batch job, the batch server shall open the standard output and standard error streams of the session.

The attributes of a batch job may indicate that the batch server executing the job shall send mail to a list of users at the time it begins execution of the job.

3.1.7 Batch Job Exit

When the session leader of an executing job terminates, the job exits. As part of exiting a batch job, the batch server that manages the job shall remove the job from the batch queue in which it resides. The server shall transfer output files of the job to a location described by the attributes of the job.

The attributes of a batch job may indicate that the batch server managing the job shall send mail to a list of users at the time the job exits.

3.1.8 Batch Job Abort

A batch server shall abort jobs for which a required deferred service cannot be performed. The attributes of a batch job may indicate that the batch server that aborts the job shall send mail to a list of users at the time it aborts the job.

3.1.9 Batch Authorization

Clients, such as the batch environment utilities (marked BE), access batch services by means of requests to one or more batch servers. To acquire the services of any given batch server, the user identifier under which the client runs must be authorized to use that batch server.

The user with an associated user name that creates a batch job shall own the job and can perform actions such as read, modify, delete, and move.

A user identifier of the same value at a different host need not be the same user. For example, user name *smith* at host **alpha** may or may not represent the same person as user name *smith* at host **beta**. Likewise, the same person may have access to different user names on different hosts.

An implementation may optionally provide an authorization mechanism that permits one user name to access jobs under another user name.

A process on a client host may be authorized to run processes under multiple user names at a batch server host. Where appropriate, the utilities defined in this volume of IEEE Std 1003.1-200x provide a means for a user to choose from among such user names when creating or modifying a batch job.

3.1.10 Batch Administration

The processing of a batch job by a batch server is affected by the attributes of the job. The processing of a batch job may also be affected by the attributes of the batch queue in which the job resides and by the status of the batch server that manages the job. See also the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.43, Batch Administrator and the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.58, Batch Operator.

3.1.11 Batch Notification

Whereas batch servers are persistent entities, clients are often transient. For example, the *qsub* utility creates a batch job and exits. For this reason, batch servers notify users of batch job events by sending mail to the user that owns the job, or to other designated users.

3.2 Batch Services

The presence of Batch Environment option services is indicated by the configuration variable POSIX2_PBS. A conforming batch server provides services as defined in this section.

A batch server shall provide batch services in two ways:

1. The batch server provides a service at the request of a client.
2. The batch server provides a deferred service as a result of a change in conditions monitored by the batch server.

If a batch server cannot complete a request, it shall reject the request. If a batch server cannot complete a deferred service for a batch job, the batch server shall abort the batch job. Table 3-2 (on page 2307) is a summary of environment variables that shall be supported by an implementation of the batch server and utilities.

Table 3-2 Environment Variable Summary

Variable	Description
PBS_DPREFIX	Defines the directive prefix (see *qsub*)
PBS_ENVIRONMENT	Batch Job is batch or interactive (see Section 3.2.2.1 (on page 2308))
PBS_JOBID	The *job_identifier* attribute of job (see Section 3.2.3.8 (on page 2320))
PBS_JOBNAME	The *job_name* attribute of job (see Section 3.2.3.8 (on page 2320))
PBS_O_HOME	Defines the *HOME* of the batch client (see *qsub*)
PBS_O_HOST	Defines the host name of the batch client (see *qsub*)
PBS_O_LANG	Defines the *LANG* of the batch client (see *qsub*)
PBS_O_LOGNAME	Defines the *LOGNAME* of the batch client (see *qsub*)
PBS_O_MAIL	Defines the *MAIL* of the batch client (see *qsub*)
PBS_O_PATH	Defines the *PATH* of the batch client (see *qsub*)
PBS_O_QUEUE	Defines the submit queue of the batch client (see *qsub*)
PBS_O_SHELL	Defines the *SHELL* of the batch client (see *qsub*)
PBS_O_TZ	Defines the *TZ* of the batch client (see *qsub*)
PBS_O_WORKDIR	Defines the working directory of the batch client (see *qsub*)
PBS_QUEUE	Defines the initial execution queue (see Section 3.2.2.1 (on page 2308))

3.2.1 Batch Job States

A batch job shall always be in one of the following states: QUEUED, RUNNING, HELD, WAITING, EXITING, or TRANSITING. The state of a batch job determines the types of requests that the batch server that manages the batch job can accept for the batch job. A batch server shall change the state of a batch job either in response to service requests from clients or as a result of deferred services, such as job execution or job routing.

A batch job that is in the QUEUED state resides in a queue but is still pending either execution or routing, depending on the queue type.

A batch server that queues a batch job in a routing queue shall put the batch job in the QUEUED state. A batch server that puts a batch job in an execution queue, but has not yet executed the batch job, shall put the batch job in the QUEUED state. A batch job that resides in an execution queue and is executing is defined to be in the RUNNING state. While a batch job is in the RUNNING state, a session leader is associated with the batch job.

A batch job that resides in an execution queue, but is ineligible to run because of a hold attribute, is defined to be in the HELD state.

A batch job that is not held, but must wait until a future date and time before executing, is defined to be in the WAITING state.

When the session leader associated with a running job exits, the batch job shall be placed in the EXITING state.

A batch job for which the session leader has terminated is defined to be in the EXITING state, and the batch server that manages such a batch job cannot accept job modification requests that affect the batch job. While a batch job is in the EXITING state, the batch server that manages the batch job is staging output files and notifying clients of job completion. Once a batch job has exited, it no longer exists as an object managed by a batch server.

A batch job that is being moved from a routing queue to another queue is defined to be in the TRANSITING state.

When a batch job in a routing queue has been selected to be moved to a new destination, then the batch job shal be in either the QUEUED state or the TRANSITING state, depending on the batch server implementation.

Batch jobs with either a *Execution_Time* attribute value set in the future or a *Hold_Types* attribute of value not equal to NO_HOLD, or both, may be routed or held in the routing queue. The treatment of jobs with the *Execution_Time* or *Hold_Types* attributes in a routing queue is implementation-defined.

When a batch job in a routing queue has not been selected to be moved to a new destination and the batch job has a *Hold_Types* attribute value of other than NO_HOLD, then the job should be in the HELD state.

Note: The effect of a hold upon a batch job in a routing queue is implementation-defined. The implementation should use the state that matches whether the batch job can route with a hold or not.

When a batch job in a routing queue has not been selected to be moved to a new destination and the batch job has:

- A *Hold_Types* attribute value of NO_HOLD
- An *Execution_Time* attribute in the past

then the batch job shall be in the QUEUED state.

When a batch job in a routing queue has not been selected to be moved to a new destination and the batch job has:

- A *Hold_Types* attribute value of NO_HOLD
- An *Execution_Time* attribute in the future

then the batch job may be in the WAITING state.

Note: The effect of a future execution time upon a batch job in a routing queue is implementation-defined. The implementation should use the state that matches whether the batch job can route with a hold or not.

Table 3-3 (on page 2309) describes the next state of a batch job, given the current state of the batch job and the type of request. Table 3-4 (on page 2310) describes the response of a batch server to a request, given the current state of the batch job and the type of request.

3.2.2 Deferred Batch Services

This section describes the deferred services performed by batch servers: job execution, job routing, job exit, job abort, and the rerunning of jobs after a restart.

3.2.2.1 Batch Job Execution

To execute a batch job is to create a session leader (a process) that runs the shell program indicated by the *Shell_Path_List* attribute of the batch job. The script is passed to the program as its standard input. An implementation may pass the script to the program by other implementation-defined means. At the time a batch job begins execution, it is defined to enter the RUNNING state.

Table 3-3 Next State Table

Request Type	**Current State**						
	X	**Q**	**R**	**H**	**W**	**E**	**T**
Queue Batch Job Request	Q	e	e	e	e	e	e
Modify Batch Job Request	e	Q	R	H	W	e	T
Delete Batch Job Request	e	X	E	X	X	E	X
Batch Job Message Request	e	Q	R	H	W	E	T
Rerun Batch Job Request	e	e	Q	e	e	e	e
Signal Batch Job Request	e	e	R	H	W	e	e
Batch Job Status Request	e	Q	R	H	W	E	T
Batch Queue Status Request	X	Q	R	H	W	E	T
Server Status Request	X	Q	R	H	W	E	T
Select Batch Jobs Request	X	Q	R	H	W	E	T
Move Batch Job Request	e	Q	R	H	W	e	T
Hold Batch Job Request	e	H	R / H	H	H	e	T
Release Batch Job Request	Q	R	Q / W / H	W	e	T	
Server Shutdown Request	X	Q	Q	H	W	E	T
Locate Batch Job Request	e	Q	R	H	W	E	T

Legend

X	Nonexistent
Q	QUEUED
R	RUNNING
H	HELD
W	WAITING
E	EXITING
T	TRANSITING
e	Error

A batch server that has an execution queue containing jobs is said to own the queue and manage the batch jobs in that queue. A batch server that has been started shall execute the batch jobs in the execution queues owned by the batch server. The batch server shall schedule for execution those jobs in the execution queues that are in the QUEUED state. The algorithm for scheduling jobs is implementation-defined.

A batch server that executes a batch job shall create, in the environment of the session leader of the batch job, an environment variable named *PBS_ENVIRONMENT*, the value of which is the string PBS_BATCH encoded in the portable character set.

A batch server that executes a batch job shall create, in the environment of the session leader of the batch job, an environment variable named *PBS_QUEUE*, the value of which is the name of the execution queue of the batch job encoded in the portable character set.

To rerun a batch job is to requeue a batch job that is currently executing and then kill the session leader of the executing job by sending a SIGKILL prior to completion; see Section 3.2.3.11 (on page 2322). A batch server that reruns a batch job shall append the standard output and standard error files of the batch job to the corresponding files of the previous execution, if they exist, with appropriate annotation. If either file does not exist, that file shall be created as in normal execution.

Table 3-4 Results / Output Table

Request Type	Current State						
	X	**Q**	**R**	**H**	**W**	**E**	**T**
Queue Batch Job Request	O	e	e	e	e	e	e
Modify Batch Job Request	e	O	e	O	O	e	e
Delete Batch Job Request	e	O	O	O	O	e	O
Batch Job Message Request	e	e	O	e	e	e	e
Rerun Batch Job Request	e	e	O	e	e	e	e
Signal Batch Job Request	e	e	O	e	e	e	e
Batch Job Status Request	e	O	O	O	O	O	O
Batch Queue Status Request	O	O	O	O	O	O	O
Server Status Request	O	O	O	O	O	O	O
Select Batch Job Request	e	O	O	O	O	O	O
Move Batch Job Request	e	O	O	O	O	e	e
Hold Batch Job Request	e	O	O	O	O	e	e
Release Batch Job Request	e	O	e	O	O	e	e
Server Shutdown Request	O	O	e	O	O	e	e
Locate Batch Job Request	e	O	O	O	O	O	O

Legend

O	OK
e	Error message

The execution of a batch job by a batch server shall be controlled by job, queue, and server attributes, as defined in this section.

Account_Name Attribute

Batch accounting is an optional feature of batch servers. If a batch server implements accounting, the statements in this section apply and the configuration variable POSIX2_PBS_ACCOUNTING shall be set to 1.

A batch server that executes a batch job shall charge the account named in the *Account_Name* attribute of the batch job for resources consumed by the batch job.

If the *Account_Name* attribute of the batch job is absent from the batch job attribute list or is altered while the batch job is in execution, the batch server action is implementation-defined.

Checkpoint Attribute

Batch checkpointing is an optional feature of batch servers. If a batch server implements checkpointing, the statements in this section apply and the configuration variable POSIX2_PBS_CHECKPOINT shall be set to 1.

There are two attributes associated with the checkpointing feature: *Checkpoint* and *Minimum_Cpu_Interval*. *Checkpoint* is a batch job attribute, while *Minimum_Cpu_Interval* is a queue attribute. An implementation that does not support checkpointing shall support the *Checkpoint* job attribute to the extent that the batch server shall maintain and pass this attribute to other servers.

The behavior of a batch server that executes a batch job for which the value of the *Checkpoint* attribute is CHECKPOINT_UNSPECIFIED is implementation-defined. A batch server that executes a batch job for which the value of the *Checkpoint* attribute is NO_CHECKPOINT shall not checkpoint the batch job.

A batch server that executes a batch job for which the value of the *Checkpoint* attribute is CHECKPOINT_AT_SHUTDOWN shall checkpoint the batch job only when the batch server accepts a request to shut down during the time when the batch job is in the RUNNING state.

A batch server that executes a batch job for which the value of the *Checkpoint* attribute is CHECKPOINT_AT_MIN_CPU_INTERVAL shall checkpoint the batch job at the interval specified by the *Minimum_Cpu_Interval* attribute of the queue for which the batch job has been selected. The *Minimum_Cpu_Interval* attribute shall be specified in units of CPU minutes.

A batch server that executes a batch job for which the value of the *Checkpoint* attribute is an unsigned integer shall checkpoint the batch job at an interval that is the value of either the *Checkpoint* attribute, or the *Minimum_Cpu_Interval* attribute of the queue for which the batch job has been selected, whichever is greater. Both intervals shall be in units of CPU minutes. When the *Minimum_Cpu_Interval* attribute is greater than the *Checkpoint* attribute, the batch job shall write a warning message to the standard error stream of the batch job.

Error_Path Attribute

The *Error_Path* attribute of a running job cannot be changed by a *Modify Batch Job Request*. When the *Join_Path* attribute of the batch job is set to the value FALSE and the *Keep_Files* attribute of the batch job does not contain the value KEEP_STD_ERROR, a batch server that executes a batch job shall perform one of the following actions:

- Set the standard error stream of the session leader of the batch job to the path described by the value of the *Error_Path* attribute of the batch job.
- Buffer the standard error of the session leader of the batch job until completion of the batch job, and when the batch job exits return the contents to the destination described by the value of the *Error_Path* attribute of the batch job.

Applications shall not rely on having access to the standard error of a batch job prior to the completion of the batch job.

When the *Error_Path* attribute does not specify a host name, then the batch server shall retain the standard error of the batch job on the host of execution.

When the *Error_Path* attribute does specify a host name and the *Keep_Files* attribute does not contain the value KEEP_STD_ERROR, then the final destination of the standard error of the batch job shall be on the host whose host name is specified.

If the path indicated by the value of the *Error_Path* attribute of the batch job is a relative path, the batch server shall expand the path relative to the home directory of the user on the host to which the file is being returned.

When the batch server buffers the standard error of the batch job and the file cannot be opened for write upon completion of the batch job, then the server shall place the standard error in an implementation-defined location and notify the user of the location via mail. It shall be possible for the user to process this mail using the *mailx* utility.

If a batch server that does not buffer the standard error cannot open the standard error path of the batch job for write access, then the batch server shall abort the batch job.

Execution_Time Attribute

A batch server shall not execute a batch job before the time represented by the value of the *Execution_Time* attribute of the batch job. The *Execution_Time* attribute is defined in seconds since the Epoch.

Hold_Types Attribute

A batch server shall support the following hold types:

`s`	Can be set or released by a user with at least a privilege level of batch administrator (SYSTEM).
`o`	Can be set or released by a user with at least a privilege level of batch operator (OPERATOR).
`u`	Can be set or released by the user with at least a privilege level of user, where the user is defined in the *Job_Owner* attribute (USER).
`n`	Indicates that none of the *Hold_Types* attributes are set (NO_HOLD).

An implementation may define other hold types. Any additional hold types, how they are specified, their internal representation, their behavior, and how they affect the behavior of other utilities are implementation-defined.

The value of the *Hold_Types* attribute shall be the union of the valid hold types (`'s'`, `'o'`, `'u'`, and any implementation-defined hold types), or `'n'`.

A batch server shall not execute a batch job if the *Hold_Types* attribute of the batch job has a value other than NO_HOLD. If the *Hold_Types* attribute of the batch job has a value other than NO_HOLD, the batch job shall be in the HELD state.

Job_Owner Attribute

The *Job_Owner* attribute consists of a pair of user name and host name values of the form:

```
username@hostname
```

A batch server that accepts a *Queue Batch Job Request* shall set the *Job_Owner* attribute to a string that is the *username@hostname* of the user who submitted the job.

Join_Path Attribute

A batch server that executes a batch job for which the value of the *Join_Path* attribute is TRUE shall ignore the value of the *Error_Path* attribute and merge the standard error of the batch job with the standard output of the batch job.

Keep_Files Attribute

A batch server that executes a batch job for which the value of the *Keep_Files* attribute includes the value KEEP_STD_OUTPUT shall retain the standard output of the batch job on the host where execution occurs. The standard output shall be retained in the home directory of the user under whose user ID the batch job is executed and the filename shall be the default filename for the standard output as defined under the **−o** option of the *qsub* utility. The *Output_Path* attribute is not modified.

A batch server that executes a batch job for which the value of the *Keep_Files* attribute includes the value KEEP_STD_ERROR shall retain the standard error of the batch job on the host where execution occurs. The standard error shall be retained in the home directory of the user under whose user ID the batch job is executed and the filename shall be the default filename for standard error as defined under the **–e** option of the *qsub* utility. The *Error_Path* attribute is not modified.

A batch server that executes a batch job for which the value of the *Keep_Files* attribute includes values other than KEEP_STD_OUTPUT and KEEP_STD_ERROR shall retain these other files on the host where execution occurs. These files (with implementation-defined names) shall be retained in the home directory of the user under whose user identifier the batch job is executed.

Mail_Points and Mail_Users Attributes

A batch server that executes a batch job for which one of the values of the *Mail_Points* attribute is the value MAIL_AT_BEGINNING shall send a mail message to each user account listed in the *Mail_Users* attribute of the batch job.

The mail message shall contain at least the batch job identifier, queue, and server at which the batch job currently resides, and the *Job_Owner* attribute.

Output_Path Attribute

The *Output_Path* attribute of a running job cannot be changed by a *Modify Batch Job Request*. When the *Keep_Files* attribute of the batch job does not contain the value KEEP_STD_OUTPUT, a batch server that executes a batch job shall either:

- Set the standard output stream of the session leader of the batch job to the destination described by the value of the *Output_Path* attribute of the batch job.

  or:

- Buffer the standard output of the session leader of the batch job until completion of the batch job, and when the batch job exits return the contents to the destination described by the value of the *Output_Path* attribute of the batch job.

When the *Output_Path* attribute does not specify a host name, then the batch server shall retain the standard output of the batch job on the host of execution.

When the *Keep_Files* attribute does not contain the value KEEP_STD_OUTPUT and the *Output_Path* attribute does specify a host name, then the final destination of the standard output of the batch job shall be on the host specified.

If the path specified in the *Output_Path* attribute of the batch job is a relative path, the batch server shall expand the path relative to the home directory of the user on the host to which the file is being returned.

Whether or not the batch server buffers the standard output of the batch job until completion of the batch job is implementation-defined. Applications shall not rely on having access to the standard output of a batch job prior to the completion of the batch job.

When the batch server does buffer the standard output of the batch job and the file cannot be opened for write upon completion of the batch job, then the batch server shall place the standard output in an implementation-defined location and notify the user of the location via mail. It shall be possible for the user to process this mail using the *mailx* utility.

If a batch server that does not buffer the standard output cannot open the standard output path of the batch job for write access, then the batch server shall abort the batch job.

Priority Attribute

A batch server implementation may choose to preferentially execute a batch job based on the *Priority* attribute. The interpretation of the batch job *Priority* attribute by a batch server is implementation-defined. If an implementation uses the *Priority* attribute, it shall interpret larger values of the *Priority* attribute to mean the batch job shall be preferentially selected for execution.

Rerunable Attribute

A batch job that began execution but did not complete, because the batch server either shut down or terminated abnormally, shall be requeued if the *Rerunable* attribute of the batch job has the value TRUE.

If a batch job, which was requeued after beginning execution but prior to completion, has a valid checkpoint file and the batch server supports checkpointing, then the batch job shall be restarted from the last valid checkpoint.

If the batch job cannot be restarted from a checkpoint, then when a batch job has a *Rerunable* attribute value of TRUE and was requeued after beginning execution but prior to completion, the batch server shall place the batch job into execution at the beginning of the job.

When a batch job has a *Rerunable* attribute value other than TRUE and was requeued after beginning execution but prior to completion, and the batch job cannot be restarted from a checkpoint, then the batch server shall abort the batch job.

Resource_List Attribute

A batch server that executes a batch job shall establish the resource limits of the session leader of the batch job according to the values of the *Resource_List* attribute of the batch job. Resource limits shall be enforced by an implementation-defined method.

Shell_Path_List Attribute

The *Shell_Path_List* job attribute consists of a list of pairs of pathname and host name values. The host name component can be omitted, in which case the pathname serves as the default pathname when a batch server cannot find the name of the host on which it is running in the list.

A batch server that executes a batch job shall select, from the value of the *Shell_Path_List* attribute of the batch job, a pathname where the shell to execute the batch job shall be found. The batch server shall select the pathname, in order of preference, according to the following methods:

- Select the pathname that contains the name of the host on which the batch server is running.
- Select the pathname for which the host name has been omitted.
- Select the pathname for the login shell of the user under which the batch job is to execute.

If the shell path value selected is an invalid pathname, the batch server shall abort the batch job.

If the value of the selected pathname from the *Shell_Path_List* attribute of the batch job represents a partial path, the batch server shall expand the path relative to a path that is implementation-defined.

The batch server that executes the batch job shall execute the program that was selected from the *Shell_Path_List* attribute of the batch job. The batch server shall pass the path to the script of the batch job as the first argument to the shell program.

User_List Attribute

The *User_List* job attribute consists of a list of pairs of user name and host name values. The host name component can be omitted, in which case the user name serves as a default when a batch server cannot find the name of the host on which it is running in the list.

A batch server that executes a batch job shall select, from the value of the *User_List* attribute of the batch job, a user name under which to create the session leader. The server shall select the user name, in order of preference, according to the following methods:

- Select the user name of a value that contains the name of the host on which the batch server executes.
- Select the user name of a value for which the host name has been omitted.
- Select the user name from the *Job_Owner* attribute of the batch job.

Variable_List Attribute

A batch server that executes a batch job shall create, in the environment of the session leader of the batch job, each environment variable listed in the *Variable_List* attribute of the batch job, and set the value of each such environment variable to that of the corresponding variable in the variable list.

3.2.2.2 Batch Job Routing

To route a batch job is to select a queue from a list and move the batch job to that queue.

A batch server that has routing queues, which have been started, shall route the jobs in the routing queues owned by the batch server. A batch server may delay the routing of a batch job. The algorithm for selecting a batch job and the queue to which it will be routed is implementation-defined.

When a routing queue has multiple possible destinations specified, then the precedence of the destinations is implementation-defined.

A batch server that routes a batch job to a queue at another server shall move the batch job into the target queue with a *Queue Batch Job Request*.

If the target server rejects the *Queue Batch Job Request*, the routing server shall retry routing the batch job or abort the batch job. A batch server that retries failed routings shall provide a means for the batch administrator to specify the number of retries and the minimum period of time between retries. The means by which an administrator specifies the number of retries and the delay between retries is implementation-defined. When the number of retries specified by the batch administrator has been exhausted, the batch server shall abort the batch job and perform the functions of *Batch Job Exit*; see Section 3.2.2.3.

3.2.2.3 Batch Job Exit

For each job in the EXITING state, the batch server that exited the batch job shall perform the following deferred services in the order specified:

1. If buffering standard error, move that file into the location specified by the *Error_Path* attribute of the batch job.
2. If buffering standard output, move that file into the location specified by the *Output_Path* attribute of the batch job.
3. If the *Mail_Points* attribute of the batch job includes MAIL_AT_EXIT, send mail to the users listed in the *Mail_Users* attribute of the batch job. The mail message shall contain at least

the batch job identifier, queue, and server at which the batch job currently resides, and the *Job_Owner* attribute.

4. Remove the batch job from the queue.

If a batch server that buffers the standard error output cannot return the standard error file to the standard error path at the time the batch job exits, the batch server shall do one of the following:

- Mail the standard error file to the batch job owner.
- Save the standard error file and mail the location and name of the file where the standard error is stored to the batch job owner.
- Save the standard error file and notify the user by other implementation-defined means.

If a batch server that buffers the standard output cannot return the standard output file to the standard output path at the time the batch job exits, the batch server shall do one of the following:

- Mail the standard output file to the batch job owner.
- Save the standard output file and mail the location and name of the file where the standard output is stored to the batch job owner.
- Save the standard output file and notify the user by other implementation-defined means.

At the conclusion of job exit processing, the batch job is no longer managed by a batch server.

3.2.2.4 *Batch Server Restart*

A batch server that has been either shutdown or terminated abnormally, and has returned to operation, is said to have *restarted*.

Upon restarting, a batch server shall requeue those jobs managed by the batch server that were in the RUNNING state at the time the batch server shut down and for which the *Rerunable* attribute of the batch job has the value TRUE.

Queues are defined to be non-volatile. A batch server shall store the content of queues that it controls in such a way that server and system shutdowns do not erase the content of the queues.

3.2.2.5 *Batch Job Abort*

A batch server that cannot perform a deferred service for a batch job shall abort the batch job.

A batch server that aborts a batch job shall perform the following services:

- Delete the batch job from the queue in which it resides.
- If the *Mail_Points* attribute of the batch job includes the value MAIL_AT_ABORT, send mail to the users listed in the value of the *Mail_Users* attribute of the job. The mail message shall contain at least the batch job identifier, queue, and server at which the batch job currently resides, the *Job_Owner* attribute, and the reason for the abort.
- If the batch job was in the RUNNING state, terminate the session leader of the executing job by sending the session leader a SIGKILL, place the batch job in the EXITING state, and perform the actions of *Batch Job Exit*.

3.2.3 Requested Batch Services

This section describes the services provided by batch servers in response to requests from clients. Table 3-5 summarizes the current set of batch service requests and for each gives its type (deferred or not) and whether it is an optional function.

Table 3-5 Batch Services Summary

Batch Service	**Deferred**	**Optional**
Batch Job Execution	Yes	No
Batch Job Routing	Yes	No
Batch Job Exit	Yes	No
Batch Server Restart	Yes	No
Batch Job Abort	Yes	No
Delete Batch Job Request	No	No
Hold Batch Job Request	No	No
Batch Job Message Request	No	Yes
Batch Job Status Request	No	No
Locate Batch Job Request	No	Yes
Modify Batch Job Request	No	No
Move Batch Job Request	No	No
Queue Batch Job Request	No	No
Batch Queue Status Request	No	No
Release Batch Job Request	No	No
Rerun Batch Job Request	No	No
Select Batch Jobs Request	No	No
Server Shutdown Request	No	No
Server Status Request	No	No
Signal Batch Job Request	No	No
Track Batch Job Request	No	Yes

If a request is rejected because the batch client is not authorized to perform the action, the batch server shall return the same status as when the batch job does not exist.

3.2.3.1 Delete Batch Job Request

A batch job is defined to have been deleted when it has been removed from the queue in which it resides and not instantiated in another queue. A client requests that the server that manages a batch job delete the batch job. Such a request is called a *Delete Batch Job Request*.

A batch server shall reject a *Delete Batch Job Request* if any of the following statements are true:

- The user of the batch client is not authorized to delete the designated job.
- The designated job is not managed by the batch server.
- The designated job is in a state inconsistent with the delete request.

A batch server may reject a *Delete Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server requested to delete a batch job shall delete the batch job if the batch job exists and is not in the EXITING state.

A batch server that deletes a batch job in the RUNNING state shall send a SIGKILL signal to the session leader of the batch job. It is implementation-defined whether additional signals are sent to the session leader of the job prior to sending the SIGKILL signal.

A batch server that deletes a batch job in the RUNNING state shall place the batch job in the EXITING state after it has killed the session leader of the batch job and shall perform the actions of *Batch Job Exit*.

3.2.3.2 *Hold Batch Job Request*

A batch client can request that the batch server add one or more holds to a batch job. Such a request is called a *Hold Batch Job Request*.

A batch server shall reject a *Hold Batch Job Request* if any of the following statements are true:

- The batch server does not support one or more of the requested holds to be added to the batch job.
- The user of the batch client is not authorized to add one or more of the requested holds to the batch job.
- The batch server does not manage the specified job.
- The designated job is in the EXITING state.

A batch server may reject a *Hold Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Hold Batch Job Request* for a batch job in the RUNNING state shall place a hold on the batch job. The effects, if any, the hold will have on a batch job in the RUNNING state are implementation-defined.

A batch server that accepts a *Hold Batch Job Request* shall add each type of hold listed in the *Hold Batch Job Request*, that is not already present, to the value of the *Hold_Types* attribute of the batch job.

3.2.3.3 *Batch Job Message Request*

Batch Job Message Request is an optional feature of batch servers. If an implementation supports *Batch Job Message Request*, the statements in this section apply and the configuration variable POSIX2_PBS_MESSAGE shall be set to 1.

A batch client can request that a batch server write a message into certain output files of a batch job. Such a request is called a *Batch Job Message Request*.

A batch server shall reject a *Batch Job Message Request* if any of the following statements are true:

- The batch server does not support sending messages to jobs.
- The user of the batch client is not authorized to post a message to the designated job.
- The designated job does not exist on the batch server.
- The designated job is not in the RUNNING state.

A batch server may reject a *Batch Job Message Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Batch Job Message Request* shall write the message sent by the batch client into the files indicated by the batch client.

3.2.3.4 Batch Job Status Request

A batch client can request that a batch server respond with the status and attributes of a batch job. Such a request is called a *Batch Job Status Request*.

A batch server shall reject a *Batch Job Status Request* if any of the following statements are true:

- The user of the batch client is not authorized to query the status of the designated job.
- The designated job is not managed by the batch server.

A batch server may reject a *Batch Job Status Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Batch Job Status Request* shall return a *Batch Job Status Message* to the batch client.

A batch server may return other information in response to a *Batch Job Status Request*.

3.2.3.5 Locate Batch Job Request

Locate Batch Job Request is an optional feature of batch servers. If an implementation supports *Locate Batch Job Request*, the statements in this section apply and the configuration variable POSIX2_PBS_LOCATE shall be set to 1.

A batch client can ask a batch server to respond with the location of a batch job that was created by the batch server. Such a request is called a *Locate Batch Job Request*.

A batch server that accepts a *Locate Batch Job Request* shall return a *Batch Job Location Message* to the batch client.

A batch server may reject a *Locate Batch Job Request* for a batch job that was not created by that server.

A batch server may reject a *Locate Batch Job Request* for a batch job that is no longer managed by that server; that is, for a batch job that is not in a queue owned by that server.

A batch server may reject a *Locate Batch Job Request* for other implementation-defined reasons.

3.2.3.6 Modify Batch Job Request

Batch clients modify (alter) the attributes of a batch job by making a request to the server that manages the batch job. Such a request is called a *Modify Batch Job Request*.

A batch server shall reject a *Modify Batch Job Request* if any of the following statements are true:

- The user of the batch client is not authorized to make the requested modification to the batch job.
- The designated job is not managed by the batch server.
- The requested modification is inconsistent with the state of the batch job.
- An unrecognized resource is requested for a batch job in an execution queue.

A batch server may reject a *Modify Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Modify Batch Job Request* shall modify all the specified attributes of the batch job. A batch server that rejects a *Modify Batch Job Request* shall modify none of the attributes of the batch job.

If the servicing by a batch server of an otherwise valid request would result in no change, then the batch server shall indicate successful completion of the request.

3.2.3.7 Move Batch Job Request

A batch client can request that a batch server move a batch job to another destination. Such a request is called a *Move Batch Job Request*.

A batch server shall reject a *Move Batch Job Request* if any of the following statements are true:

- The user of the batch client is not authorized to remove the designated job from the queue in which the batch job resides.
- The user of the batch client is not authorized to move the designated job to the destination.
- The designated job is not managed by the batch server.
- The designated job is in the EXITING state.
- The destination is inaccessible.

A batch server can reject a *Move Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Move Batch Job Request* shall perform the following services:

- Queue the designated job at the destination.
- Remove the designated job from the queue in which the batch job resides.

If the destination resides on another batch server, the batch server shall queue the batch job at the destination by sending a *Queue Batch Job Request* to the other server. If the *Queue Batch Job Request* fails, the batch server shall reject the *Move Batch Job Request*. If the *Queue Batch Job Request* succeeds, the batch server shall remove the batch job from its queue.

The batch server shall not modify any attributes of the batch job.

3.2.3.8 Queue Batch Job Request

A batch queue is controlled by one and only one batch server. A batch server is said to own the queues that it controls. Batch clients make requests of batch servers to have jobs queued. Such a request is called a *Queue Batch Job Request*.

A batch server requested to queue a batch job for which the queue is not specified shall select an implementation-defined queue for the batch job. Such a queue is called the *default queue* of the batch server. The implementation shall provide the means for a batch administrator to specify the default queue. The queue, whether specified or defaulted, is called the *target queue.*

A batch server shall reject a *Queue Batch Job Request* if any of the following statements are true:

- The client is not authorized to create a batch job in the target queue.
- The request specifies a queue that does not exist on the batch server.
- The target queue is an execution queue and the batch server cannot satisfy a resource requirement of the batch job.
- The target queue is an execution queue and an unrecognized resource is requested.
- The target queue is an execution queue, the batch server does not support checkpointing, and the value of the *Checkpoint* attribute of the batch job is not NO_CHECKPOINT.

- The job requires access to a user identifier that the batch client is not authorized to access.

A batch server may reject a *Queue Batch Job Request* for other implementation-defined reasons.

A batch server that accepts a *Queue Batch Job Request* for a batch job for which the PBS_O_QUEUE value is missing from the value of the *Variable_List* attribute of the batch job shall add that variable to the list and set the value to the name of the target queue. Once set, no server shall change the value of PBS_O_QUEUE, even if the batch job is moved to another queue.

A batch server that accepts a *Queue Batch Job Request* for a batch job for which the PBS_JOBID value is missing from the value of the *Variable_List* attribute shall add that variable to the list and set the value to the batch job identifier assigned by the server in the format:

```
sequence_number.server
```

A batch server that accepts a *Queue Batch Job Request* for a batch job for which the PBS_JOBNAME value is missing from the value of the *Variable_List* attribute of the batch job shall add that variable to the list and set the value to the *Job_Name* attribute of the batch job.

3.2.3.9 Batch Queue Status Request

A batch client can request that a batch server respond with the status and attributes of a queue. Such a request is called a *Batch Queue Status Request*.

A batch server shall reject a *Batch Queue Status Request* if any of the following statements are true:

- The user of the batch client is not authorized to query the status of the designated queue.
- The designated queue does not exist on the batch server.

A batch server may reject a *Batch Queue Status Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Batch Queue Status Request* shall return a *Batch Queue Status Reply* to the batch client.

3.2.3.10 Release Batch Job Request

A batch client can request that server remove one or more holds from a batch job. Such a request is called a *Release Batch Job Request*.

A batch server shall reject a *Release Batch Job Request* if any of the following statements are true:

- The user of the batch client is not authorized to remove one or more of the requested holds from the batch job.
- The batch server does not manage the specified job.

A batch server may reject a *Release Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Release Batch Job Request* shall remove each type of hold listed in the *Release Batch Job Request*, that is present, from the value of the *Hold_Types* attribute of the batch job.

3.2.3.11 *Rerun Batch Job Request*

To rerun a batch job is to kill the session leader of the batch job and leave the batch job eligible for re-execution. A batch client can request that a batch server rerun a batch job. Such a request is called *Rerun Batch Job Request*.

A batch server shall reject a *Rerun Batch Job Request* if any of the following statements are true:

- The user of the batch client is not authorized to rerun the designated job.
- The *Rerunable* attribute of the designated job has the value FALSE.
- The designated job is not in the RUNNING state.
- The batch server does not manage the designated job.

A batch server may reject a *Rerun Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that rejects a *Rerun Batch Job Request* shall in no way modify the execution of the batch job.

A batch server that accepts a request to rerun a batch job shall perform the following services:

- Requeue the batch job in the execution queue in which it was executing.
- Send a SIGKILL signal to the process group of the session leader of the batch job.

An implementation may indicate to the batch job owner that the batch job has been rerun. Whether and how the batch job owner is notified that a batch job is rerun is implementation-defined.

A batch server that reruns a batch job may send other implementation-defined signals to the session leader of the batch job prior to sending the SIGKILL signal.

A batch server may preferentially select a rerun job for execution. Whether rerun jobs shall be selected for execution before other jobs is implementation-defined.

3.2.3.12 *Select Batch Jobs Request*

A batch client can request from a batch server a list of jobs managed by that server that match a list of selection criteria. Such a request is called a *Select Batch Jobs Request*. All the batch jobs managed by the batch server that receives the request are candidates for selection.

A batch server that accepts a *Select Batch Jobs Request* shall return a list of zero or more job identifiers that correspond to jobs that meet the selection criteria.

If the batch client is not authorized to query the status of a batch job, the batch server shall not select the batch job.

3.2.3.13 *Server Shutdown Request*

A batch server is defined to have shut down when it does not respond to requests from clients and does not perform deferred services for jobs. A batch client can request that a batch server shut down. Such a request is called a *Server Shutdown Request*.

A batch server shall reject a *Server Shutdown Request* from a client that is not authorized to shut down the batch server. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server may reject a *Server Shutdown Request* for other implementation-defined reasons. The reasons for which a *Server Shutdown Request* may be rejected are implementation-defined.

At server shutdown, a batch server shall do, in order of preference, one of the following:

- If checkpointing is implemented and the batch job is checkpointable, then checkpoint the batch job and requeue it.
- If the batch job is rerunable, then requeue the batch job to be rerun (restarted from the beginning).
- Abort the batch job.

3.2.3.14 *Server Status Request*

A batch client can request that a batch server respond with the status and attributes of the batch server. Such a request is called a *Server Status Request*.

A batch server shall reject a *Server Status Request* if the following statement is true:

- The user of the batch client is not authorized to query the status of the designated server.

A batch server may reject a *Server Status Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a *Server Status Request* shall return a *Server Status Reply* to the batch client.

3.2.3.15 *Signal Batch Job Request*

A batch client can request that a batch server signal the session leader of a batch job. Such a request is called a *Signal Batch Job Request*.

A batch server shall reject a *Signal Batch Job Request* if any of the following statements are true:

- The user of the batch client is not authorized to signal the batch job.
- The job is not in the RUNNING state.
- The batch server does not manage the designated job.
- The requested signal is not supported by the implementation.

A batch server may reject a *Signal Batch Job Request* for other implementation-defined reasons. The method used to determine whether the user of a client is authorized to perform the requested action is implementation-defined.

A batch server that accepts a request to signal a batch job shall send the signal requested by the batch client to the process group of the session leader of the batch job.

3.2.3.16 *Track Batch Job Request*

Track Batch Job Request is an optional feature of batch servers. If an implementation supports *Track Batch Job Request*, the statements in this section apply and the configuration variable POSIX2_PBS_TRACK shall be set to 1.

Track Batch Job Request provides a method for tracking the current location of a batch job. Clients may use the tracking information to determine the batch server that should receive a batch server request.

If *Track Batch Job Request* is supported by a batch server, then when the batch server queues a batch job as a result of a *Queue Batch Job Request*, and the batch server is not the batch server that created the batch job, the batch server shall send a *Track Batch Job Request* to the batch server that created the job.

If *Track Batch Job Request* is supported by a batch server, then the *Track Batch Job Request* may also be sent to other servers as a backup to the primary server. The method by which backup servers are specified is implementation-defined.

If *Track Batch Job Request* is supported by a batch server that receives a *Track Batch Job Request*, then the batch server shall record the current location of the batch job as contained in the request.

3.3 Common Behavior for Batch Environment Utilities

3.3.1 Batch Job Identifier

A utility shall recognize *job_identifier*s of the format:

```
[sequence_number][.server_name][@server]
```

where:

sequence_number	An integer that, when combined with *server_name*, provides a batch job identifier that is unique within the batch system.
server_name	The name of the batch server to which the batch job was originally submitted.
server	The name of the batch server that is currently managing the batch job.

If the application omits the batch *server_name* portion of a batch job identifier, a utility shall use the name of a default batch server.

If the application omits the batch *server* portion of a batch job identifier, a utility shall use:

- The batch server indicated by *server_name*, if present.
- The name of the default batch server.
- The name of the batch server that is currently managing the batch job.

If only *@server* is specified, then the status of all jobs owned by the user on the requested server is listed.

The means by which a utility determines the default batch server is implementation-defined.

If the application presents the batch *server* portion of a batch job identifier to a utility, the utility shall send the request to the specified server.

A strictly conforming application shall use the syntax described for the job identifier. Whenever a batch job identifier is specified whose syntax is not recognized by an implementation, then a message for each error that occurs shall be written to standard error and the utility shall exit with an exit status greater than zero.

When a batch job identifier is supplied as an argument to a batch utility and the *server_name* portion of the batch job identifier is omitted, then the utility shall use the name of the default batch server.

When a batch job identifier is supplied as an argument to a batch utility and the batch *server* portion of the batch job identifier is omitted, then the utility shall use either:

- The name of the default batch server

  or:

- The name of the batch server that is currently managing the batch job

When a batch job identifier is supplied as an argument to a batch utility and the batch *server* portion of the batch job identifier is specified, then the utility shall send the required *Batch Server Request* to the specified server.

3.3.2 Destination

The utility shall recognize a *destination* of the format:

```
[queue][@server]
```

where:

queue	The name of a valid execution or routing queue at the batch server denoted by *@server*, defined as a string of up to 15 alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set) where the first character is alphabetic.
server	The name of a batch server, defined as a string of alphanumeric characters in the portable character set.

If the application omits the batch *server* portion of a destination, then the utility shall use either:

- The name of the default batch server

  or:

- The name of the batch server that is currently managing the batch job

The means by which a utility determines the default batch server is implementation-defined.

If the application omits the *queue* portion of a destination, then the utility shall use the name of the default queue at the batch server chosen. The means by which a batch server determines its default queue is implementation-defined. If a destination is specified in the *queue@server* form, then the utility shall use the specified queue at the specified server.

A strictly conforming application shall use the syntax described for a destination. Whenever a destination is specified whose syntax is not recognized by an implementation, then a message shall be written to standard error and the utility shall exit with an exit status greater than zero.

3.3.3 Multiple Keyword-Value Pairs

For each option that can have multiple keyword-value pair arguments, the following rules shall apply. Examples of options that can have list-oriented option-arguments are −**u** *value@keyword* and −**l** *keyword=value*.

1. If a batch utility is presented with a list-oriented option-argument for which a keyword has a corresponding value that begins with a single or double quote, then the utility shall stop interpreting the input stream for delimiters until a second single or double quote, respectively, is encountered. This feature allows some flexibility for a comma (`','`) or equals sign (`'='`) to be part of the value string for a particular keyword; for example:

   ```
   keywd1='val1,val2',keywd2="val3,val4"
   ```

   Note: This may require the user to escape the quotes as in the following command:

```
foo −xkeywd1=\'val1,val2\',keywd2=\"val3,val4\"
```

2. If a batch server is presented with a list-oriented attribute that has a keyword that was encountered earlier in the list, then the later entry for that keyword shall replace the earlier entry.

3. If a batch server is presented with a list-oriented attribute that has a keyword without any corresponding value of the form *keyword*= or @*keyword* and the same keyword was encountered earlier in the list, then the prior entry for that keyword shall be ignored by the batch server.

4. If a batch utility is expecting a list-oriented option-argument entry of the form *keyword*=*value*, but is presented with an entry of the form *keyword* without any corresponding *value*, then the entry shall be treated as though a default value of NULL was assigned (that is, *keyword*=NULL) for entry parsing purposes. The utility shall include only the keyword, not the NULL value, in the associated job attribute.

5. If a batch utility is expecting a list-oriented option-argument entry of the form *value*@*keyword*, but is presented with an entry of the form *value* without any corresponding *keyword*, then the entry shall be treated as though a keyword of NULL was assigned (that is, *value*@NULL) for entry parsing purposes. The utility shall include only the value, not the NULL keyword, in the associated job attribute.

6. A batch server shall accept a list-oriented attribute that has multiple occurrences of the same keyword, interpreting the keywords, in order, with the last value encountered taking precedence over prior instances of the same keyword. This rule allows, but does not require, a batch utility to preprocess the attribute to remove duplicate keywords.

7. If a batch utility is presented with multiple list-oriented option-arguments on the command line or in script directives, or both, for a single option, then the utility shall concatenate, in order, any command line keyword and value pairs to the end of any directive keyword and value pairs separated by a single comma to produce a single string that is an equivalent, valid option-argument. The resulting string shall be assigned to the associated attribute of the batch job (after optionally removing duplicate entries as described in item 6.

Chapter 4

Utilities

This chapter contains the definitions of the utilities, as follows:

- Mandatory utilities that are present on every conformant system
- Optional utilities that are present only on systems supporting the associated option; see Section 1.8.1 (on page 2210) for information on the options in this volume of IEEE Std 1003.1-200x

NAME

admin — create and administer SCCS files (**DEVELOPMENT**)

SYNOPSIS

XSI

```
admin -i[name][-n][-a login][-d flag][-e login][-f flag][-m mrlist]
    [-r rel][-t[name][-y[comment]] newfile
```

```
admin -n[-a login][-d flag][-e login][-f flag][-m mrlist][-t[name]]
    [-y[comment]] newfile ...
```

```
admin [-a login][-d flag][-m mrlist][-r rel][-t[name]] file ...
```

```
admin -h file ...
```

```
admin -z file ...
```

DESCRIPTION

The *admin* utility shall create new SCCS files or change parameters of existing ones. If a named file does not exist, it shall be created, and its parameters shall be initialized according to the specified options. Parameters not initialized by an option shall be assigned a default value. If a named file does exist, parameters corresponding to specified options shall be changed, and other parameters shall be left as is.

All SCCS filenames supplied by the application shall be of the form s.*filename*. New SCCS files shall be given read-only permission mode. Write permission in the parent directory is required to create a file. All writing done by *admin* shall be to a temporary *x-file*, named x.*filename* (see *get*) created with read-only mode if *admin* is creating a new SCCS file, or created with the same mode as that of the SCCS file if the file already exists. After successful execution of *admin*, the SCCS file shall be removed (if it exists), and the *x-file* shall be renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occur.

The *admin* utility shall also use a transient lock file (named z.*filename*), which is used to prevent simultaneous updates to the SCCS file; see *get* (on page 2675).

OPTIONS

The *admin* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the **–i**, **–t**, and **–y** options have optional option-arguments. These optional option-arguments shall not be presented as separate arguments. The following options are supported:

–n Create a new SCCS file. When **–n** is used without **–i**, the SCCS file shall be created with control information but without any file data.

–i[*name***]** Specify the *name* of a file from which the text for a new SCCS file shall be taken. The text constitutes the first delta of the file (see the **–r** option for delta numbering scheme). If the **–i** option is used, but the *name* option-argument is omitted, the text shall be obtained by reading the standard input. If this option is omitted, the SCCS file shall be created with control information but without any file data. The **–i** option implies the **–n** option.

–r *SID* Specify the SID of the initial delta to be inserted. This SID shall be a trunk SID; that is, the branch and sequence numbers shall be zero or missing. The level number is optional, and defaults to 1.

–t[*name***]** Specify the *name* of a file from which descriptive text for the SCCS file shall be taken. In the case of existing SCCS files (neither **–i** nor **–n** is specified):

- A **−t** option without a *name* option-argument shall cause the removal of descriptive text (if any) currently in the SCCS file.
- A **−t** option with a *name* option-argument shall cause the text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

−f *flag* Specify a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **−f** options may be supplied on a single *admin* command line. Implementations shall recognize the following flags and associated values:

b Allow use of the **−b** option on a *get* command to create branch deltas.

c*ceil* Specify the highest release (that is, ceiling), a number less than or equal to 9 999, which may be retrieved by a *get* command for editing. The default value for an unspecified **c** flag shall be 9 999.

f*floor* Specify the lowest release (that is, floor), a number greater than 0 but less than 9 999, which may be retrieved by a *get* command for editing. The default value for an unspecified **f** flag shall be 1.

d*SID* Specify the default delta number (SID) to be used by a *get* command.

i*str* Treat the ‘‘No ID keywords’’ message issued by *get* or *delta* as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get* (on page 2675)) are found in the text retrieved or stored in the SCCS file. If a value is supplied, the application shall ensure that the keywords exactly match the given string; however, the string shall contain a keyword, and no embedded <newline>s.

j Allow concurrent *get* commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* Specify a *list* of releases to which deltas can no longer be made (that is, *get* **−e** against one of these locked releases fails). Conforming applications shall use the following syntax to specify a *list*. Implementations may accept additional forms as an extension:

```
<list> ::= a | <range-list>
<range-list> ::= <range> | <range-list>, <range>
<range> ::= <SID>
```

The character *a* in the *list* shall be equivalent to specifying all releases for the named SCCS file. The non-terminal <*SID*> in range shall be the delta number of an existing delta associated with the SCCS file.

n Cause *delta* to create a null delta in each of those releases (if any) being skipped when a delta is made in a new release (for example, in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas shall serve as anchor points so that branch deltas may later be created from them. The absence of this flag shall cause skipped releases to be nonexistent in the SCCS file, preventing branch deltas from being created from them in the future. During the initial creation of an SCCS file, the **n** flag may be ignored; that is, if the **−r** option is used to set the release number of the initial SID to a value greater than 1, null deltas need not be created for the ‘‘skipped’’ releases.

q*text* Substitute user-definable *text* for all occurrences of the %**Q**% keyword in the SCCS file text retrieved by *get*.

m*mod* Specify the module name of the SCCS file substituted for all occurrences of the %**M**% keyword in the SCCS file text retrieved by *get*. If the **m** flag is not specified, the value assigned shall be the name of the SCCS file with the leading `'.'` removed.

t*type* Specify the *type* of module in the SCCS file substituted for all occurrences of the %**Y**% keyword in the SCCS file text retrieved by *get*.

v*pgm* Cause *delta* to prompt for modification request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validation program. (If this flag is set when creating an SCCS file, the application shall ensure that the **m** option is also used even if its value is null.)

−**d** *flag* Remove (delete) the specified *flag* from an SCCS file. Several −**d** options may be supplied on a single *admin* command. See the −**f** option for allowable *flag* names. (The **l***list* flag gives a *list* of releases to be unlocked. See the −**f** option for further description of the **l** flag and the syntax of a *list*.)

−**a** *login* Specify a *login* name, or numerical group ID, to be added to the list of users who may make deltas (changes) to the SCCS file. A group ID shall be equivalent to specifying all *login* names common to that group ID. Several −**a** options may be used on a single *admin* command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a `'!'`, the users so specified shall be denied permission to make deltas.

−**e** *login* Specify a *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several −**e** options may be used on a single *admin* command line.

−**y**[*comment*] Insert the *comment* text into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*. In the POSIX locale, omission of the −**y** option shall result in a default comment line being inserted in the form:

```
"date and time created %s %s by %s", <date>, <time>, <login>
```

where <*date*> is expressed in the format of the *date* utility's `%y/%m/%d` conversion specification, <*time*> in the format of the *date* utility's `%T` conversion specification format, and <*login*> is the login name of the user creating the file.

−**m** *mrlist* Insert the list of modification request (MR) numbers into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*. The application shall ensure that the **v** flag is set and the MR numbers are validated if the **v** flag has a value (the name of an MR number validation program). A diagnostic message shall be written if the **v** flag is not set or MR validation fails.

−**h** Check the structure of the SCCS file and compare the newly computed checksum (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. If the newly computed checksum does not match the checksum in the SCCS file, a diagnostic message shall be written.

−z	Recompute the SCCS file checksum and store it in the first line of the SCCS file (see the **−h** option above). Note that use of this option on a truly corrupted file may prevent future detection of the corruption.

OPERANDS

The following operands shall be supported:

file	A pathname of an existing SCCS file or a directory. If *file* is a directory, the *admin* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files shall be silently ignored.
newfile	A pathname of an SCCS file to be created.

If exactly one *file* or *newfile* operand appears, and it is `'−'`, the standard input shall be read; each line of the standard input shall be taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only if the **−i** is specified without an option-argument or if a *file* or *newfile* operand is specified as `'−'`. If the first character of any standard input line is <SOH> in the POSIX locale, the results are unspecified.

INPUT FILES

The existing SCCS files shall be text files of an unspecified format.

The application shall ensure that the file named by the **−i** option's *name* option-argument shall be a text file; if the first character of any line in this file is <SOH> in the POSIX locale, the results are unspecified. If this file contains more than 99 999 lines, the number of lines recorded in the header for this file shall be 99 999 for this delta.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *admin*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and the contents of the default **−y** comment.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Any SCCS files created shall be text files of an unspecified format. During processing of a *file*, a locking *z-file*, as described in *get* (on page 2675), may be created and deleted.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

It is recommended that directories containing SCCS files be writable by the owner only, and that SCCS files themselves be read-only. The mode of the directories should allow only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delta, *get*, *prs*, *what*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

The grammar is updated.

The Open Group Base Resolution bwg 2001-007 is applied, adding new text to the INPUT FILES section warning that the maximum lines recorded in the file is 99 999.

NAME

alias — define or display aliases

SYNOPSIS

UP
```
alias [alias-name[=string] ...]
```

DESCRIPTION

The *alias* utility shall create or redefine alias definitions or write the values of existing alias definitions to standard output. An alias definition provides a string value that shall replace a command name when it is encountered; see Section 2.3.1 (on page 2234).

An alias definition shall affect the current shell execution environment and the execution environments of the subshells of the current shell. When used as specified by this volume of IEEE Std 1003.1-200x, the alias definition shall not affect the parent process of the current shell nor any utility environment invoked by the shell; see Section 2.12 (on page 2263).

OPTIONS

None.

OPERANDS

The following operands shall be supported:

alias-name	Write the alias definition to standard output.
alias-name=string	Assign the value of *string* to the alias *alias-name*.

If no operands are given, all alias definitions shall be written to standard output.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *alias*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The format for displaying aliases (when no operands or only *name* operands are specified) shall be:

```
"%s=%s\n", name, value
```

The *value* string shall be written with appropriate quoting so that it is suitable for reinput to the shell. See the description of shell quoting in Section 2.2 (on page 2232).

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	One of the *name* operands specified did not have an alias definition, or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. Change *ls* to give a columnated, more annotated output:

```
alias ls="ls −CF"
```

2. Create a simple "redo" command to repeat previous entries in the command history file:

```
alias r='fc −s'
```

3. Use 1K units for *du*:

```
alias du=du\ −k
```

4. Set up *nohup* so that it can deal with an argument that is itself an alias name:

```
alias nohup="nohup "
```

RATIONALE

The *alias* description is based on historical KornShell implementations. Known differences exist between that and the C shell. The KornShell version was adopted to be consistent with all the other KornShell features in this volume of IEEE Std 1003.1-200x, such as command line editing.

Since *alias* affects the current shell execution environment, it is generally provided as a shell regular built-in.

Historical versions of the KornShell have allowed aliases to be exported to scripts that are invoked by the same shell. This is triggered by the *alias* −**x** flag; it is allowed by this volume of IEEE Std 1003.1-200x only when an explicit extension such as −**x** is used. The standard developers considered that aliases were of use primarily to interactive users and that they should normally not affect shell scripts called by those users; functions are available to such scripts.

Historical versions of the KornShell had not written aliases in a quoted manner suitable for reentry to the shell, but this volume of IEEE Std 1003.1-200x has made this a requirement for all similar output. Therefore, consistency with this volume of IEEE Std 1003.1-200x was chosen over this detail of historical practice.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.9.5 (on page 2256)

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

NAME

ar — create and maintain library archives

SYNOPSIS

```
SD    ar -d[-v] archive file ...

XSI   ar -m[-abiv][posname] archive file ...

XSI   ar -p[-v][-s]archive [file ...]
XSI   ar -q[-cv] archive file ...

XSI   ar -r[-cuv][-abi][posname]archive file ...
XSI   ar -t[-v][-s]archive [file ...]
XSI   ar -x[-v][-sCT]archive [file ...]
```

DESCRIPTION

The *ar* utility can be used to create and maintain groups of files combined into an archive. Once an archive has been created, new files can be added, and existing files in an archive can be extracted, deleted, or replaced. When an archive consists entirely of valid object files, the implementation shall format the archive so that it is usable as a library for link editing (see *c99* and *fort77*). When some of the archived files are not valid object files, the suitability of the archive for library use is undefined. XSI If an archive consists entirely of printable files, the entire archive shall be printable.

When *ar* creates an archive, it creates administrative information indicating whether a symbol table is present in the archive. When there is at least one object file that *ar* recognizes as such in the archive, an archive symbol table shall be created in the archive and maintained by *ar*; it is used by the link editor to search the archive. Whenever the *ar* utility is used to create or update the contents of such an archive, the symbol table shall be rebuilt. The **–s** option shall force the symbol table to be rebuilt.

All *file* operands can be pathnames. However, files within archives shall be named by a filename, which is the last component of the pathname used when the file was entered into the archive. The comparison of *file* operands to the names of files in archives shall be performed by comparing the last component of the operand to the name of the file in the archive.

It is unspecified whether multiple files in the archive may be identically named. In the case of such files, however, each *file* XSI and *posname* operand shall match only the first file in the archive having a name that is the same as the last component of the operand.

OPTIONS

The *ar* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

XSI	**–a**	Position new files in the archive after the file named by the *posname* operand.
XSI	**–b**	Position new files in the archive before the file named by the *posname* operand.
	–c	Suppress the diagnostic message that is written to standard error by default when the archive *archive* is created.
XSI	**–C**	Prevent extracted files from replacing like-named files in the file system. This option is useful when **–T** is also used, to prevent truncated filenames from

replacing files with the same prefix.

−d Delete one or more *file*s from *archive*.

XSI **−i** Position new files in the archive before the file in the archive named by the *posname* operand (equivalent to **−b**).

XSI **−m** Move the named files in the archive. The **−a**, **−b**, or **−i** options with the *posname* operand indicate the position; otherwise, move the names files in the archive to the end of the archive.

−p Write the contents of the *file*s in the archive named by *file* operands from *archive* to the standard output. If no *file* operands are specified, the contents of all files in the archive shall be written in the order of the archive.

XSI **−q** Append the named files to the end of the archive. In this case *ar* does not check whether the added files are already in the archive. This is useful to bypass the searching otherwise done when creating a large archive piece by piece.

−r Replace or add *file*s to *archive*. If the archive named by *archive* does not exist, a new archive shall be created and a diagnostic message shall be written to standard error (unless the **−c** option is specified). If no *file*s are specified and the *archive* exists, the results are undefined. Files that replace existing files in the archive shall not change the order of the archive. Files that do not replace existing files in the archive shall be appended to the archive unless a **−a**, **−b**, or **−i** option specifies another position.

XSI **−s** Force the regeneration of the archive symbol table even if *ar* is not invoked with an option that modifies the archive contents. This option is useful to restore the archive symbol table after it has been stripped; see *strip*.

−t Write a table of contents of *archive* to the standard output. The files specified by the *file* operands shall be included in the written list. If no *file* operands are specified, all files in *archive* shall be included in the order of the archive.

XSI **−T** Allow filename truncation of extracted files whose archive names are longer than the file system can support. By default, extracting a file with a name that is too long shall be an error; a diagnostic message shall be written and the file shall not be extracted.

−u Update older files in the archive. When used with the **−r** option, files in the archive shall be replaced only if the corresponding *file* has a modification time that is at least as new as the modification time of the file in the archive.

−v Give verbose output. When used with the option characters **−d**, **−r**, or **−x**, write a detailed file-by-file description of the archive creation and maintenance activity, as described in the STDOUT section.

When used with **−p**, write the name of the file in the archive to the standard output before writing the file in the archive itself to the standard output, as described in the STDOUT section.

When used with **−t**, include a long listing of information about the files in the archive, as described in the STDOUT section.

−x Extract the files in the archive named by the *file* operands from *archive*. The contents of the archive shall not be changed. If no *file* operands are given, all files in the archive shall be extracted. The modification time of each file extracted shall be set to the time the file is extracted from the archive.

OPERANDS

The following operands shall be supported:

archive	A pathname of the archive.
file	A pathname. Only the last component shall be used when comparing against the names of files in the archive. If two or more *file* operands have the same last pathname component (basename), the results are unspecified. The implementation's archive format shall not truncate valid filenames of files added to or replaced in the archive.
XSI *posname*	The name of a file in the archive, used for relative positioning; see options −**m** and −**r**.

STDIN

Not used.

INPUT FILES

The archive named by *archive* shall be a file in the format created by *ar* −**r**.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ar*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_TIME	Determine the format and content for date and time strings written by *ar* −**tv**.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
TMPDIR	Determine the pathname that overrides the default directory for temporary files, if any.
TZ	Determine the timezone used to calculate date and time strings written by *ar* −**tv**. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the −**d** option is used with the −**v** option, the standard output format shall be:

```
"d − %s\n", <file>
```

where *file* is the operand specified on the command line.

If the −**p** option is used with the −**v** option, *ar* shall precede the contents of each file with:

```
"\n<%s>\n\n", <file>
```

where *file* is the operand specified on the command line, if *file* operands were specified, and the name of the file in the archive if they were not.

If the **−r** option is used with the **−v** option:

- If *file* is already in the archive, the standard output format shall be:

  ```
  "r − %s\n", <file>
  ```

  where <*file*> is the operand specified on the command line.

- If *file* is not already in the archive, the standard output format shall be:

  ```
  "a − %s\n", <file>
  ```

  where <*file*> is the operand specified on the command line.

If the **−t** option is used, *ar* shall write the names of the files in the archive to the standard output in the format:

```
"%s\n", <file>
```

where *file* is the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

If the **−t** option is used with the **−v** option, the standard output format shall be:

```
"%s %u/%u %u %s %d %d:%d %d %s\n", <member mode>, <user ID>,
    <group ID>, <number of bytes in member>,
    <abbreviated month>, <day-of-month>, <hour>,
    <minute>, <year>, <file>
```

where:

<*file*>	Shall be the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.
<*member mode*>	Shall be formatted the same as the <*file mode*> string defined in the STDOUT section of *ls*, except that the first character, the <*entry type*>, is not used; the string represents the file mode of the file in the archive at the time it was added to or replaced in the archive.

The following represent the last-modification time of a file when it was most recently added to or replaced in the archive:

<*abbreviated month*>	Equivalent to the format of the `%b` conversion specification format in *date*.
<*day-of-month*>	Equivalent to the format of the `%e` conversion specification format in *date*.
<*hour*>	Equivalent to the format of the `%H` conversion specification format in *date*.
<*minute*>	Equivalent to the format of the `%M` conversion specification format in *date*.
<*year*>	Equivalent to the format of the `%Y` conversion specification format in *date*.

When *LC_TIME* does not specify the POSIX locale, a different format and order of presentation of these fields relative to each other may be used in a format appropriate in the specified locale.

If the −**x** option is used with the −**v** option, the standard output format shall be:

```
"x − %s\n", <file>
```

where *file* is the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

STDERR

The standard error shall be used only for diagnostic messages. The diagnostic message about creating a new archive when −**c** is not specified shall not modify the exit status.

OUTPUT FILES

Archives are files with unspecified formats.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The archive format is not described. It is recognized that there are several known *ar* formats, which are not compatible. The *ar* utility is included, however, to allow creation of archives that are intended for use only on one machine. The archive is specified as a file, and it can be moved as a file. This does allow an archive to be moved from one machine to another machine that uses the same implementation of *ar*.

Utilities such as *pax* (and its forebears *tar* and *cpio*) also provide portable ‘‘archives’’. This is a not a duplication; the *ar* utility is included to provide an interface primarily for *make* and the compilers, based on a historical model.

In historical implementations, the −**q** option (available on XSI-conforming systems) is known to execute quickly because *ar* does not check on whether the added members are already in the archive. This is useful to bypass the searching otherwise done when creating a large archive piece-by-piece. These remarks may but need not remain true for a brand new implementation of this utility; hence, these remarks have been moved into the RATIONALE.

BSD implementations historically required applications to provide the −**s** option whenever the archive was supposed to contain a symbol table. As in this volume of IEEE Std 1003.1-200x, System V historically creates or updates an archive symbol table whenever an object file is removed from, added to, or updated in the archive.

The OPERANDS section requires what might seem to be true without specifying it: the archive cannot truncate the filenames below {NAME_MAX}. Some historical implementations do so, however, causing unexpected results for the application. Therefore, this volume of IEEE Std 1003.1-200x makes the requirement explicit to avoid misunderstandings.

According to the System V documentation, the options −**dmpqrtx** are not required to begin with a hyphen (`'−'`). This volume of IEEE Std 1003.1-200x requires that a conforming application use the leading hyphen.

The archive format used by the 4.4 BSD implementation is documented in this RATIONALE as an example:

> A file created by *ar* begins with the ''magic'' string `"!<arch>\n"`. The rest of the archive is made up of objects, each of which is composed of a header for a file, a possible filename, and the file contents. The header is portable between machine architectures, and, if the file contents are printable, the archive is itself printable.
>
> The header is made up of six ASCII fields, followed by a two-character trailer. The fields are the object name (16 characters), the file last modification time (12 characters), the user and group IDs (each 6 characters), the file mode (8 characters), and the file size (10 characters). All numeric fields are in decimal, except for the file mode, which is in octal.
>
> The modification time is the file *st_mtime* field. The user and group IDs are the file *st_uid* and *st_gid* fields. The file mode is the file *st_mode* field. The file size is the file *st_size* field. The two-byte trailer is the string `"<newline>"`.
>
> Only the name field has any provision for overflow. If any filename is more than 16 characters in length or contains an embedded space, the string `"#1/"` followed by the ASCII length of the name is written in the name field. The file size (stored in the archive header) is incremented by the length of the name. The name is then written immediately following the archive header.
>
> Any unused characters in any of these fields are written as <space>s. If any fields are their particular maximum number of characters in length, there is no separation between the fields.
>
> Objects in the archive are always an even number of bytes long; files that are an odd number of bytes long are padded with a <newline>, although the size in the header does not reflect this.

The *ar* utility description requires that (when all its members are valid object files) *ar* produce an object code library, which the linkage editor can use to extract object modules. If the linkage editor needs a symbol table to permit random access to the archive, *ar* must provide it; however, *ar* does not require a symbol table.

The BSD −**o** option was omitted. It is a rare conforming application that uses *ar* to extract object code from a library with concern for its modification time, since this can only be of importance to *make*. Hence, since this functionality is not deemed important for applications portability, the modification time of the extracted files is set to the current time.

There is at least one known implementation (for a small computer) that can accommodate only object files for that system, disallowing mixed object and other files. The ability to handle any type of file is not only historical practice for most implementations, but is also a reasonable expectation.

Consideration was given to changing the output format of *ar* −**tv** to the same format as the output of *ls* −**l**. This would have made parsing the output of *ar* the same as that of *ls*. This was rejected in part because the current *ar* format is commonly used and changes would break historical usage. Second, *ar* gives the user ID and group ID in numeric format separated by a slash. Changing this to be the user name and group name would not be correct if the archive were moved to a machine that contained a different user database. Since *ar* cannot know whether the archive was generated on the same machine, it cannot tell what to report.

The text on the **–ur** option combination is historical practice—since one filename can easily represent two different files (for example, **/a/foo** and **/b/foo**), it is reasonable to replace the file in the archive even when the modification time in the archive is identical to that in the file system.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *pax*, *strip* the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers, **<unistd.h>** description of {POSIX_NO_TRUNC}

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the Software Development Utilities option.

The STDOUT description is changed for the **–v** option to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

IEEE PASC Interpretation 1003.2 #198 is applied, changing the description to consistently use ‘‘file’’ to refer to a file in the file system hierarchy, ‘‘archive’’ to refer to the archive being operated upon by the *ar* utility, and ‘‘file in the archive’’ to refer to a copy of a file that is contained in the archive.

NAME

asa — interpret carriage-control characters

SYNOPSIS

FR `asa [ file ... ]`

DESCRIPTION

The *asa* utility shall write its input files to standard output, mapping carriage-control characters from the text files to line-printer control sequences in an implementation-defined manner.

The first character of every line shall be removed from the input, and the following actions are performed:

If the character removed is:

<space>	The rest of the line is output without change.
0	A <newline> is output, then the rest of the input line.
1	One or more implementation-defined characters that causes an advance to the next page shall be output, followed by the rest of the input line.
+	The <newline> of the previous line shall be replaced with one or more implementation-defined characters that causes printing to return to column position 1, followed by the rest of the input line. If the `'+'` is the first character in the input, it shall be equivalent to <space>. |

The action of the *asa* utility is unspecified upon encountering any character other than those listed above as the first character in a line.

OPTIONS

None.

OPERANDS

file	A pathname of a text file used for input. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified; see the INPUT FILES section. |

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *asa*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be the text from the input file modified as described in the DESCRIPTION section.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were output successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. The following command:

   ```
   asa file
   ```

   permits the viewing of *file* (created by a program using FORTRAN-style carriage control characters) on a terminal.

2. The following command:

   ```
   a.out | asa | lp
   ```

   formats the FORTRAN output of **a.out** and directs it to the printer.

RATIONALE

The *asa* utility is needed to map ‘‘standard’’ FORTRAN 77 output into a form acceptable to contemporary printers. Usually, *asa* is used to pipe data to the *lp* utility; see *lp*.

This utility is generally used only by FORTRAN programs. The standard developers decided to retain *asa* to avoid breaking the historical large base of FORTRAN applications that put carriage-control characters in their output files. There is no requirement that a system have a FORTRAN compiler in order to run applications that need *asa*.

Historical implementations have used an ASCII <form-feed> in response to a 1 and an ASCII <carriage-return> in response to a `'+'`. It is suggested that implementations treat characters other than 0, 1, and `'+'` as <space> in the absence of any compelling reason to do otherwise. However, the action is listed here as ‘‘unspecified’’, permitting an implementation to provide extensions to access fast multiple-line slewing and channel seeking in a non-portable manner.

FUTURE DIRECTIONS

None.

SEE ALSO

fort77, *lp*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the FORTRAN Runtime Utilities option.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

at — execute commands at a later time

SYNOPSIS

UP

```
at [−m][−f file][−q queuename] −t time_arg

at [−m][−f file][−q queuename] timespec ...

at −r at_job_id ...

at −l −q queuename

at −l [at_job_id ...]
```

DESCRIPTION

The *at* utility shall read commands from standard input and group them together as an *at-job*, to be executed at a later time.

The at-job shall be executed in a separate invocation of the shell, running in a separate process group with no controlling terminal, except that the environment variables, current working directory, file creation mask, and other implementation-defined execution-time attributes in effect when the *at* utility is executed shall be retained and used when the at-job is executed.

When the at-job is submitted, the *at_job_id* and scheduled time shall be written to standard error. The *at_job_id* is an identifier that shall be a string consisting solely of alphanumeric characters and the period character. The *at_job_id* shall be assigned by the system when the job is scheduled such that it uniquely identifies a particular job.

User notification and the processing of the job's standard output and standard error are described under the −**m** option.

XSI Users shall be permitted to use *at* if their name appears in the file **/usr/lib/cron/at.allow**. If that file does not exist, the file **/usr/lib/cron/at.deny** shall be checked to determine whether the user shall be denied access to *at*. If neither file exists, only a process with the appropriate privileges shall be allowed to submit a job. If only **at.deny** exists and is empty, global usage shall be permitted. The **at.allow** and **at.deny** files shall consist of one user name per line.

OPTIONS

The *at* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−**f** *file* Specify the pathname of a file to be used as the source of the at-job, instead of standard input.

−**l** (The letter ell.) Report all jobs scheduled for the invoking user if no *at_job_id* operands are specified. If *at_job_id*s are specified, report only information for these jobs. The output shall be written to standard output.

−**m** Send mail to the invoking user after the at-job has run, announcing its completion. Standard output and standard error produced by the at-job shall be mailed to the user as well, unless redirected elsewhere. Mail shall be sent even if the job produces no output.

If −**m** is not used, the job's standard output and standard error shall be provided to the user by means of mail, unless they are redirected elsewhere; if there is no such output to provide, the implementation need not notify the user of the job's completion.

−q *queuename*	Specify in which queue to schedule a job for submission. When used with the **−l** option, limit the search to that particular queue. By default, at-jobs shall be scheduled in queue *a*. In contrast, queue *b* shall be reserved for batch jobs; see *batch*. The meanings of all other *queuename*s are implementation-defined. If **−q** is specified along with either of the **−t** *time_arg* or *timespec* arguments, the results are unspecified.
−r	Remove the jobs with the specified *at_job_id* operands that were previously scheduled by the *at* utility.
−t *time_arg*	Submit the job to be run at the time specified by the *time* option-argument, which the application shall ensure has the format as specified by the *touch* **−t** *time* utility.

OPERANDS

The following operands shall be supported:

at_job_id The name reported by a previous invocation of the *at* utility at the time the job was scheduled.

timespec Submit the job to be run at the date and time specified. All of the *timespec* operands are interpreted as if they were separated by <space>s and concatenated, and shall be parsed as described in the grammar at the end of this section. The date and time shall be interpreted as being in the timezone of the user (as determined by the *TZ* variable), unless a timezone name appears as part of *time*, below.

In the POSIX locale, the following describes the three parts of the time specification string. All of the values from the *LC_TIME* categories in the POSIX locale shall be recognized in a case-insensitive manner.

time The time can be specified as one, two, or four digits. One-digit and two-digit numbers shall be taken to be hours; four-digit numbers to be hours and minutes. The time can alternatively be specified as two numbers separated by a colon, meaning *hour*:*minute*. An AM/PM indication (one of the values from the **am_pm** keywords in the *LC_TIME* locale category) can follow the time; otherwise, a 24-hour clock time shall be understood. A timezone name can also follow to further qualify the time. The acceptable timezone names are implementation-defined, except that they shall be case-insensitive and the string **utc** is supported to indicate the time is in Coordinated Universal Time. In the POSIX locale, the *time* field can also be one of the following tokens:

midnight	Indicates the time 12:00 am (00:00).
noon	Indicates the time 12:00 pm.
now	Indicates the current day and time. Invoking *at* <**now**> shall submit an at-job for potentially immediate execution (that is, subject only to unspecified scheduling delays).

date An optional *date* can be specified as either a month name (one of the values from the **mon** or **abmon** keywords in the *LC_TIME* locale category) followed by a day number (and possibly year number preceded by a comma), or a day of the week (one of the values from the **day** or **abday** keywords in the *LC_TIME* locale category). In the POSIX locale, two special days shall be recognized:

today	Indicates the current day.
tomorrow	Indicates the day following the current day.

If no *date* is given, **today** shall be assumed if the given time is greater than the current time, and **tomorrow** shall be assumed if it is less. If the given month is less than the current month (and no year is given), next year shall be assumed.

increment The optional *increment* shall be a number preceded by a plus sign ('+') and suffixed by one of the following: **minutes**, **hours**, **days**, **weeks**, **months**, or **years**. (The singular forms shall be also accepted.) The keyword **next** shall be equivalent to an increment number of +1. For example, the following are equivalent commands:

```
at 2pm + 1 week
at 2pm next week
```

The following grammar describes the precise format of *timespec* in the POSIX locale. The general conventions for this style of grammar are described in Section 1.10 (on page 2220). This formal syntax shall take precedence over the preceding text syntax description. The longest possible token or delimiter shall be recognized at a given point. When used in a *timespec*, white space shall also delimit tokens.

```
%token hr24clock_hr_min
%token hr24clock_hour
/*
  A hr24clock_hr_min is a one, two, or four-digit number. A one-digit
  or two-digit number constitutes a hr24clock_hour. A hr24clock_hour
  may be any of the single digits [0,9], or may be double digits, ranging
  from [00,23]. If a hr24clock_hr_min is a four digit number, the
  first two digits shall be a valid hr24clock_hour, while the last two
  represent the number of minutes, from [00,59].
*/

%token wallclock_hr_min
%token wallclock_hour
/*
  A wallclock_hr_min is a one, two-digit, or four-digit number.
  A one-digit or two-digit number constitutes a wallclock_hour.
  A wallclock_hour may be any of the single digits [1,9], or may
  be double digits, ranging from [01,12]. If a wallclock_hr_min
  is a four-digit number, the first two digits shall be a valid
  wallclock_hour, while the last two represent the number of
  minutes, from [00,59].
*/

%token minute
/*
  A minute is a one or two-digit number whose values can be [0,9]
  or [00,59].
*/

%token day_number
/*
  A day_number is a number in the range appropriate for the particular
  month and year specified by month_name and year_number, respectively.
```

```
      If no year_number is given, the current year is assumed if the given
      date and time are later this year. If no year_number is given and
      the date and time have already occurred this year and the month is
      not the current month, next year is the assumed year.
    */

    %token year_number
    /*
      A year_number is a four-digit number representing the year A.D., in
      which the at_job is to be run.
    */

    %token inc_number
    /*
      The inc_number is the number of times the succeeding increment
      period is to be added to the specified date and time.
    */

    %token timezone_name
    /*
      The name of an optional timezone suffix to the time field, in an
      implementation-defined format.
    */

    %token month_name
    /*
      One of the values from the mon or abmon keywords in the LC_TIME
      locale category.
    */

    %token day_of_week
    /*
      One of the values from the day or abday keywords in the LC_TIME
      locale category.
    */

    %token am_pm
    /*
      One of the values from the am_pm keyword in the LC_TIME locale
      category.
    */

    %start timespec
    %%
    timespec    : time
                | time date
                | time increment
                | time date increment
                | nowspec
                ;

    nowspec     : "now"
                | "now" increment
                ;

    time        : hr24clock_hr_min
                | hr24clock_hr_min timezone_name
```

```
                    | hr24clock_hour ":" minute
                    | hr24clock_hour ":" minute timezone_name
                    | wallclock_hr_min am_pm
                    | wallclock_hr_min am_pm timezone_name
                    | wallclock_hour ":" minute am_pm
                    | wallclock_hour ":" minute am_pm timezone_name
                    | "noon"
                    | "midnight"
                    ;

        date        : month_name day_number
                    | month_name day_number "," year_number
                    | day_of_week
                    | "today"
                    | "tomorrow"
                    ;

        increment   : "+" inc_number inc_period
                    | "next" inc_period
                    ;

        inc_period  : "minute" | "minutes"
                    | "hour" | "hours"
                    | "day" | "days"
                    | "week" | "weeks"
                    | "month" | "months"
                    | "year" | "years"
                    ;
```

STDIN

The standard input shall be a text file consisting of commands acceptable to the shell command language described in Chapter 2 (on page 2231). The standard input shall only be used if no **–f** *file* option is specified.

INPUT FILES

See the STDIN section.

XSI The text files **/usr/lib/cron/at.allow** and **/usr/lib/cron/at.deny** shall contain zero or more user names, one per line, of users who are, respectively, authorized or denied access to the *at* and *batch* utilities.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *at*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of

	diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
LC_TIME	Determine the format and contents for date and time strings written and accepted by *at*.
SHELL	Determine a name of a command interpreter to be used to invoke the at-job. If the variable is unset or null, *sh* shall be used. If it is set to a value other than a name for *sh*, the implementation shall do one of the following: use that shell; use *sh*; use the login shell from the user database; or any of the preceding accompanied by a warning diagnostic about which was chosen.
TZ	Determine the timezone. The job shall be submitted for execution at the time specified by *timespec* or **–t** *time* relative to the timezone specified by the *TZ* variable. If *timespec* specifies a timezone, it shall override *TZ*. If *timespec* does not specify a timezone and *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When standard input is a terminal, prompts of unspecified format for each line of the user input described in the STDIN section may be written to standard output.

In the POSIX locale, the following shall be written to the standard output for each job when jobs are listed in response to the **–l** option:

```
"%s\t%s\n", at_job_id, <date>
```

where *date* shall be equivalent in format to the output of:

```
date +"%a %b %e %T %Y"
```

The date and time written shall be adjusted so that they appear in the timezone of the user (as determined by the *TZ* variable).

STDERR

In the POSIX locale, the following shall be written to standard error when a job has been successfully submitted:

```
"job %s at %s\n", at_job_id, <date>
```

where *date* has the same format as is described in the STDOUT section. Neither this, nor warning messages concerning the selection of the command interpreter, shall be considered a diagnostic that changes the exit status.

Diagnostic messages, if any, shall be written to standard error.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 The *at* utility successfully submitted, removed, or listed a job or jobs.

>0 An error occurred.

CONSEQUENCES OF ERRORS

The job shall not be scheduled, removed, or listed.

APPLICATION USAGE

The format of the *at* command line shown here is guaranteed only for the POSIX locale. Other cultures may be supported with substantially different interfaces, although implementations are encouraged to provide comparable levels of functionality.

Since the commands run in a separate shell invocation, running in a separate process group with no controlling terminal, open file descriptors, traps, and priority inherited from the invoking environment are lost.

Some implementations do not allow substitution of different shells using *SHELL*. System V systems, for example, have used the login shell value for the user in **/etc/passwd**. To select reliably another command interpreter, the user must include it as part of the script, such as:

```
$ at 1800
myshell myscript
job ... at ...
$
```

EXAMPLES

1. This sequence can be used at a terminal:

```
at −m 0730 tomorrow
sort < file >outfile
EOT
```

2. This sequence, which demonstrates redirecting standard error to a pipe, is useful in a command procedure (the sequence of output redirection specifications is significant):

```
at now + 1 hour <<!
diff file1 file2 2>&1 >outfile | mailx mygroup
!
```

3. To have a job reschedule itself, *at* can be invoked from within the at-job. For example, this daily processing script named **my.daily** runs every day (although *crontab* is a more appropriate vehicle for such work):

```
# my.daily runs every day
daily processing
at now tomorrow < my.daily
```

4. The spacing of the three portions of the POSIX locale *timespec* is quite flexible as long as there are no ambiguities. Examples of various times and operand presentation include:

```
at 0815am Jan 24
at 8 :15amjan24
at now "+ 1day"
at 5 pm FRIday
at ’17
    utc+
    30minutes’
```

RATIONALE

The *at* utility reads from standard input the commands to be executed at a later time. It may be useful to redirect standard output and standard error within the specified commands.

The **–t** *time* option was added as a new capability to support an internationalized way of specifying a time for execution of the submitted job.

Early proposals added a ''jobname'' concept as a way of giving submitted jobs names that are meaningful to the user submitting them. The historical, system-specified *at_job_id* gives no indication of what the job is. Upon further reflection, it was decided that the benefit of this was not worth the change in historical interface. The *at* functionality is useful in simple environments, but in large or complex situations, the functionality provided by the Batch Services option is more suitable.

The **–q** option historically has been an undocumented option, used mainly by the *batch* utility.

The System V **–m** option was added to provide a method for informing users that an at-job had completed. Otherwise, users are only informed when output to standard error or standard output are not redirected.

The behavior of *at* <*now*> was changed in an early proposal from being unspecified to submitting a job for potentially immediate execution. Historical BSD *at* implementations support this. Historical System V implementations give an error in that case, but a change to the System V versions should have no backwards compatibility ramifications.

On BSD-based systems, a **–u** *user* option has allowed those with appropriate privileges to access the work of other users. Since this is primarily a system administration feature and is not universally implemented, it has been omitted. Similarly, a specification for the output format for user with appropriate privileges viewing the queues of other users has been omitted.

The **–f** *file* option from System V is used instead of the BSD method of using the last operand as the pathname. The BSD method is ambiguous—does:

```
at 1200 friday
```

mean the same thing if there is a file named **friday** in the current directory?

The *at_job_id* is composed of a limited character set in historical practice, and it is mandated here to invalidate systems that might try using characters that require shell quoting or that could not be easily parsed by shell scripts.

The *at* utility varies between System V and BSD systems in the way timezones are used. On System V systems, the *TZ* variable affects the at-job submission times and the times displayed for the user. On BSD systems, *TZ* is not taken into account. The BSD behavior is easily achieved with the current specification. If the user wishes to have the timezone default to that of the system, they merely need to issue the *at* command immediately following an unsetting or null assignment to *TZ*. For example:

```
TZ= at noon ...
```

gives the desired BSD result.

While the *yacc*-like grammar specified in the OPERANDS section is lexically unambiguous with respect to the digit strings, a lexical analyzer would probably be written to look for and return digit strings in those cases. The parser could then check whether the digit string returned is a valid *day_number*, *year_number*, and so on, based on the context.

FUTURE DIRECTIONS

None.

SEE ALSO

batch, *crontab*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- If −**m** is not used, the job's standard output and standard error are provided to the user by mail.

The effects of using the −**q** and −**t** options as defined in the IEEE P1003.2b draft standard are specified.

The normative text is reworded to avoid use of the term ''must'' for application requirements.

NAME

awk — pattern scanning and processing language

SYNOPSIS

```
awk [−F ERE][−v assignment] ... program [argument ...]

awk [−F ERE] −f progfile ... [−v assignment] ...[argument ...]
```

DESCRIPTION

The *awk* utility shall execute programs written in the *awk* programming language, which is specialized for textual data manipulation. An *awk* program is a sequence of patterns and corresponding actions. When input is read that matches a pattern, the action associated with that pattern is carried out.

Input shall be interpreted as a sequence of records. By default, a record is a line, less its terminating <newline>, but this can be changed by using the **RS** built-in variable. Each record of input shall be matched in turn against each pattern in the program. For each pattern matched, the associated action shall be executed.

The *awk* utility shall interpret each input record as a sequence of fields where, by default, a field is a string of non-<blank>s. This default white-space field delimiter can be changed by using the **FS** built-in variable or the **−F** *ERE*. The *awk* utility shall denote the first field in a record $1, the second $2, and so on. The symbol $0 shall refer to the entire record; setting any other field causes the re-evaluation of $0. Assigning to $0 shall reset the values of all other fields and the **NF** built-in variable.

OPTIONS

The *awk* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−F *ERE* Define the input field separator to be the extended regular expression *ERE*, before any input is read; see **Regular Expressions** (on page 2363).

−f *progfile* Specify the pathname of the file *progfile* containing an *awk* program. If multiple instances of this option are specified, the concatenation of the files specified as *progfile* in the order specified shall be the *awk* program. The *awk* program can alternatively be specified in the command line as a single argument.

−v *assignment*

The application shall ensure that the *assignment* argument is in the same form as an *assignment* operand. The specified variable assignment shall occur prior to executing the *awk* program, including the actions associated with **BEGIN** patterns (if any). Multiple occurrences of this option can be specified.

OPERANDS

The following operands shall be supported:

program If no **−f** option is specified, the first operand to *awk* shall be the text of the *awk* program. The application shall supply the *program* operand as a single argument to *awk*. If the text does not end in a <newline>, *awk* shall interpret the text as if it did.

argument Either of the following two types of *argument* can be intermixed:

file A pathname of a file that contains the input to be read, which is matched against the set of patterns in the program. If no *file* operands are specified, or if a *file* operand is `'−'`, the standard input shall be used.

assignment An operand that begins with an underscore or alphabetic character from the portable character set (see the table in the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set), followed by a sequence of underscores, digits, and alphabetics from the portable character set, followed by the `'='` character, shall specify a variable assignment rather than a pathname. The characters before the `'='` represent the name of an *awk* variable; if that name is an *awk* reserved word (see **Grammar** (on page 2372)) the behavior is undefined. The characters following the equal sign shall be interpreted as if they appeared in the *awk* program preceded and followed by a double-quote (`'"'`) character, as a **STRING** token (see **Grammar** (on page 2372)), except that if the last character is an unescaped backslash, it shall be interpreted as a literal backslash rather than as the first character of the sequence `"\""`. The variable shall be assigned the value of that **STRING** token and, if appropriate, shall be considered a *numeric string* (see **Expressions in awk** (on page 2358)), the variable shall also be assigned its numeric value. Each such variable assignment shall occur just prior to the processing of the following *file*, if any. Thus, an assignment before the first *file* argument shall be executed after the **BEGIN** actions (if any), while an assignment after the last *file* argument shall occur before the **END** actions (if any). If there are no *file* arguments, assignments shall be executed before processing the standard input.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'−'`; see the INPUT FILES section. If the *awk* program contains no actions and no patterns, but is otherwise a valid *awk* program, standard input and any *file* operands shall not be read and *awk* shall exit with a return status of zero.

INPUT FILES

Input files to the *awk* program from any of the following sources shall be text files:

- Any *file* operands or their equivalents, achieved by modifying the *awk* variables **ARGV** and **ARGC**
- Standard input in the absence of any *file* operands
- Arguments to the **getline** function

Whether the variable **RS** is set to a value other than a <newline> or not, for these files, implementations shall support records terminated with the specified separator up to {LINE_MAX} bytes and may support longer records.

If −**f** *progfile* is specified, the application shall ensure that the files named by each of the *progfile* option-arguments are text files containing an *awk* program.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *awk*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions and in comparisons of string values.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the behavior of character classes within regular expressions, the identification of characters as letters, and the mapping of uppercase and lowercase characters for the **toupper** and **tolower** functions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_NUMERIC	Determine the radix character used when interpreting numeric input, performing conversions between numeric and string values, and formatting numeric output. Regardless of locale, the period character (the decimal-point character of the POSIX locale) is the decimal-point character recognized in processing *awk* programs (including assignments in command line arguments).
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the search path when looking for commands executed by *system*(*expr*), or input and output pipes; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

In addition, all environment variables shall be visible via the *awk* variable **ENVIRON**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The nature of the output files depends on the *awk* program.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The nature of the output files depends on the *awk* program.

EXTENDED DESCRIPTION

Overall Program Structure

An *awk* program is composed of pairs of the form:

```
pattern { action }
```

Either the pattern or the action (including the enclosing brace characters) can be omitted.

A missing pattern shall match any record of input, and a missing action shall be equivalent to:

```
{ print }
```

Execution of the *awk* program shall start by first executing the actions associated with all **BEGIN** patterns in the order they occur in the program. Then each *file* operand (or standard input if no files were specified) shall be processed in turn by reading data from the file until a record separator is seen (<newline> by default). Before the first reference to a field in the record is evaluated, the record shall be split into fields, according to the rules in **Regular Expressions** (on page 2363), using the value of **FS** that was current at the time the record was read. Each pattern in the program then shall be evaluated in the order of occurrence, and the action associated with each pattern that matches the current record executed. The action for a matching pattern shall be executed before evaluating subsequent patterns. Finally, the actions associated with all **END** patterns shall be executed in the order they occur in the program.

Expressions in awk

Expressions describe computations used in *patterns* and *actions*. In the following table, valid expression operations are given in groups from highest precedence first to lowest precedence last, with equal-precedence operators grouped between horizontal lines. In expression evaluation, where the grammar is formally ambiguous, higher precedence operators shall be evaluated before lower precedence operators. In this table *expr*, *expr1*, *expr2*, and *expr3* represent any expression, while lvalue represents any entity that can be assigned to (that is, on the left side of an assignment operator). The precise syntax of expressions is given in **Grammar** (on page 2372).

Table 4-1 Expressions in Decreasing Precedence in *awk*

Syntax	Name	Type of Result	Associativity
`( expr )`	Grouping	Type of *expr*	N/A
`$expr`	Field reference	String	N/A
`++ lvalue`	Pre-increment	Numeric	N/A
`-- lvalue`	Pre-decrement	Numeric	N/A
`lvalue ++`	Post-increment	Numeric	N/A
`lvalue --`	Post-decrement	Numeric	N/A
`expr ^ expr`	Exponentiation	Numeric	Right
`! expr`	Logical not	Numeric	N/A
`+ expr`	Unary plus	Numeric	N/A
`- expr`	Unary minus	Numeric	N/A
`expr * expr`	Multiplication	Numeric	Left
`expr / expr`	Division	Numeric	Left
`expr % expr`	Modulus	Numeric	Left
`expr + expr`	Addition	Numeric	Left
`expr - expr`	Subtraction	Numeric	Left
`expr expr`	String concatenation	String	Left
`expr < expr`	Less than	Numeric	None
`expr <= expr`	Less than or equal to	Numeric	None
`expr != expr`	Not equal to	Numeric	None
`expr == expr`	Equal to	Numeric	None
`expr > expr`	Greater than	Numeric	None
`expr >= expr`	Greater than or equal to	Numeric	None
`expr ~ expr`	ERE match	Numeric	None
`expr !~ expr`	ERE non-match	Numeric	None

Syntax	**Name**	**Type of Result**	**Associativity**
`expr in array`	Array membership	Numeric	Left
`( index ) in array`	Multi-dimension array membership	Numeric	Left
`expr && expr`	Logical AND	Numeric	Left
`expr || expr`	Logical OR	Numeric	Left
`expr1 ? expr2 : expr3`	Conditional expression	Type of selected `expr2` or `expr3`	Right
`lvalue ^= expr`	Exponentiation assignment	Numeric	Right
`lvalue %= expr`	Modulus assignment	Numeric	Right
`lvalue *= expr`	Multiplication assignment	Numeric	Right
`lvalue /= expr`	Division assignment	Numeric	Right
`lvalue += expr`	Addition assignment	Numeric	Right
`lvalue −= expr`	Subtraction assignment	Numeric	Right
`lvalue = expr`	Assignment	Type of *expr*	Right

Each expression shall have either a string value, a numeric value, or both. Except as stated for specific contexts, the value of an expression shall be implicitly converted to the type needed for the context in which it is used. A string value shall be converted to a numeric value by the equivalent of the following calls to functions defined by the ISO C standard:

```
setlocale(LC_NUMERIC, "");
numeric_value = atof(string_value);
```

A numeric value that is exactly equal to the value of an integer (see Section 1.7.2 (on page 2207)) shall be converted to a string by the equivalent of a call to the **sprintf** function (see **String Functions** (on page 2369)) with the string `"%d"` as the *fmt* argument and the numeric value being converted as the first and only *expr* argument. Any other numeric value shall be converted to a string by the equivalent of a call to the **sprintf** function with the value of the variable **CONVFMT** as the *fmt* argument and the numeric value being converted as the first and only *expr* argument. The result of the conversion is unspecified if the value of **CONVFMT** is not a floating-point format specification. This volume of IEEE Std 1003.1-200x specifies no explicit conversions between numbers and strings. An application can force an expression to be treated as a number by adding zero to it, or can force it to be treated as a string by concatenating the null string (`""`) to it.

A string value shall be considered a *numeric string* if it comes from one of the following:

1. Field variables
2. Input from the *getline*( ) function
3. **FILENAME**
4. **ARGV** array elements
5. **ENVIRON** array elements
6. Array elements created by the *split*( ) function
7. A command line variable assignment
8. Variable assignment from another numeric string variable

and after all the following conversions have been applied, the resulting string would lexically be recognized as a **NUMBER** token as described by the lexical conventions in **Grammar** (on page

2372):

- All leading and trailing <blank>s are discarded
- If the first non-<blank> is `'+'` or `'-'`, it is discarded
- Changing each occurrence of the decimal point character from the current locale to a period

If a `'−'` character is ignored in the preceding description, the numeric value of the *numeric string* shall be the negation of the numeric value of the recognized **NUMBER** token. Otherwise, the numeric value of the *numeric string* shall be the numeric value of the recognized **NUMBER** token. Whether or not a string is a *numeric string* shall be relevant only in contexts where that term is used in this section.

When an expression is used in a Boolean context, if it has a numeric value, a value of zero shall be treated as false and any other value shall be treated as true. Otherwise, a string value of the null string shall be treated as false and any other value shall be treated as true. A Boolean context shall be one of the following:

- The first subexpression of a conditional expression
- An expression operated on by logical NOT, logical AND, or logical OR
- The second expression of a **for** statement
- The expression of an **if** statement
- The expression of the **while** clause in either a **while** or **do**. . .**while** statement
- An expression used as a pattern (as in Overall Program Structure)

All arithmetic shall follow the semantics of floating-point arithmetic as specified by the ISO C standard (see Section 1.7.2 (on page 2207)).

The value of the expression:

```
expr1 ^ expr2
```

shall be equivalent to the value returned by the ISO C standard function call:

```
pow(expr1, expr2)
```

The expression:

```
lvalue ^= expr
```

shall be equivalent to the ISO C standard expression:

```
lvalue = pow(lvalue, expr)
```

except that lvalue shall be evaluated only once. The value of the expression:

```
expr1 % expr2
```

shall be equivalent to the value returned by the ISO C standard function call:

```
fmod(expr1, expr2)
```

The expression:

```
lvalue %= expr
```

shall be equivalent to the ISO C standard expression:

```
lvalue = fmod(lvalue, expr)
```

except that lvalue shall be evaluated only once.

Variables and fields shall be set by the assignment statement:

```
lvalue = expression
```

and the type of *expression* shall determine the resulting variable type. The assignment includes the arithmetic assignments ("+=", "-=", "*=", "/=", "%=", "^=", "++", "--") all of which shall produce a numeric result. The left-hand side of an assignment and the target of increment and decrement operators can be one of a variable, an array with index, or a field selector.

The *awk* language supplies arrays that are used for storing numbers or strings. Arrays need not be declared. They shall initially be empty, and their sizes shall change dynamically. The subscripts, or element identifiers, are strings, providing a type of associative array capability. An array name followed by a subscript within square brackets can be used as an lvalue and thus as an expression, as described in the grammar; see **Grammar** (on page 2372). Unsubscripted array names can be used in only the following contexts:

- A parameter in a function definition or function call
- The **NAME** token following any use of the keyword **in** as specified in the grammar (see **Grammar** (on page 2372)); if the name used in this context is not an array name, the behavior is undefined

A valid array *index* shall consist of one or more comma-separated expressions, similar to the way in which multi-dimensional arrays are indexed in some programming languages. Because *awk* arrays are really one-dimensional, such a comma-separated list shall be converted to a single string by concatenating the string values of the separate expressions, each separated from the other by the value of the **SUBSEP** variable. Thus, the following two index operations shall be equivalent:

```
var[expr1, expr2, ... exprn]

var[expr1 SUBSEP expr2 SUBSEP ... SUBSEP exprn]
```

The application shall ensure that a multi-dimensioned *index* used with the **in** operator is parenthesized. The **in** operator, which tests for the existence of a particular array element, shall not cause that element to exist. Any other reference to a nonexistent array element shall automatically create it.

Comparisons (with the '<', "<=", "!=", "==", '>', and ">=" operators) shall be made numerically if both operands are numeric, if one is numeric and the other has a string value that is a numeric string, or if one is numeric and the other has the uninitialized value. Otherwise, operands shall be converted to strings as required and a string comparison shall be made using the locale-specific collation sequence. The value of the comparison expression shall be 1 if the relation is true, or 0 if the relation is false.

Variables and Special Variables

Variables can be used in an *awk* program by referencing them. With the exception of function parameters (see **User-Defined Functions** (on page 2371)), they are not explicitly declared. Function parameter names shall be local to the function; all other variable names shall be global. The same name shall not be used as both a function parameter name and as the name of a function or a special *awk* variable. The same name shall not be used both as a variable name with global scope and as the name of a function. The same name shall not be used within the same scope both as a scalar variable and as an array. Uninitialized variables, including scalar variables, array elements, and field variables, shall have an uninitialized value. An uninitialized value shall have both a numeric value of zero and a string value of the empty string. Evaluation of variables with an uninitialized value, to either string or numeric, shall be determined by the context in which they are used.

Field variables shall be designated by a `'$'` followed by a number or numerical expression. The effect of the field number *expression* evaluating to anything other than a non-negative integer is unspecified; uninitialized variables or string values need not be converted to numeric values in this context. New field variables can be created by assigning a value to them. References to nonexistent fields (that is, fields after $**NF**), shall evaluate to the uninitialized value. Such references shall not create new fields. However, assigning to a nonexistent field (for example, $(**NF**+2)=5) shall increase the value of **NF**; create any intervening fields with the uninitialized value; and cause the value of $0 to be recomputed, with the fields being separated by the value of **OFS**. Each field variable shall have a string value or an uninitialized value when created. Field variables shall have the uninitialized value when created from $0 using **FS** and the variable does not contain any characters. If appropriate, the field variable shall be considered a numeric string (see **Expressions in awk** (on page 2358)).

Implementations shall support the following other special variables that are set by *awk*:

ARGC The number of elements in the **ARGV** array.

ARGV An array of command line arguments, excluding options and the *program* argument, numbered from zero to **ARGC**–1.

The arguments in **ARGV** can be modified or added to; **ARGC** can be altered. As each input file ends, *awk* shall treat the next non-null element of **ARGV**, up to the current value of **ARGC**–1, inclusive, as the name of the next input file. Thus, setting an element of **ARGV** to null means that it shall not be treated as an input file. The name `'–'` indicates the standard input. If an argument matches the format of an *assignment* operand, this argument shall be treated as an assignment rather than a *file* argument.

CONVFMT The **printf** format for converting numbers to strings (except for output statements, where **OFMT** is used); `"%.6g"` by default.

ENVIRON An array representing the value of the environment, as described in the *exec* functions defined in the System Interfaces volume of IEEE Std 1003.1-200x. The indices of the array shall be strings consisting of the names of the environment variables, and the value of each array element shall be a string consisting of the value of that variable. If appropriate, the environment variable shall be considered a *numeric string* (see **Expressions in awk** (on page 2358)), the array element shall also have its numeric value.

In all cases where the behavior of *awk* is affected by environment variables (including the environment of any commands that *awk* executes via the **system** function or via pipeline redirections with the **print** statement, the **printf** statement, or the **getline** function), the environment used shall be the environment at the time *awk* began executing; it is implementation-defined whether any modification of **ENVIRON** affects this environment.

FILENAME A pathname of the current input file. Inside a **BEGIN** action the value is undefined. Inside an **END** action the value shall be the name of the last input file processed.

FNR The ordinal number of the current record in the current file. Inside a **BEGIN** action the value shall be zero. Inside an **END** action the value shall be the number of the last record processed in the last file processed.

FS	Input field separator regular expression; a <space> by default.
NF	The number of fields in the current record. Inside a **BEGIN** action, the use of **NF** is undefined unless a **getline** function without a *var* argument is executed previously. Inside an **END** action, **NF** shall retain the value it had for the last record read, unless a subsequent, redirected, **getline** function without a *var* argument is performed prior to entering the **END** action.
NR	The ordinal number of the current record from the start of input. Inside a **BEGIN** action the value shall be zero. Inside an **END** action the value shall be the number of the last record processed.
OFMT	The **printf** format for converting numbers to strings in output statements (see **Output Statements** (on page 2367)); `"%.6g"` by default. The result of the conversion is unspecified if the value of **OFMT** is not a floating-point format specification.
OFS	The **print** statement output field separation; <space> by default.
ORS	The **print** statement output record separator; a <newline> by default.
RLENGTH	The length of the string matched by the **match** function.
RS	The first character of the string value of **RS** shall be the input record separator; a <newline> by default. If **RS** contains more than one character, the results are unspecified. If **RS** is null, then records are separated by sequences consisting of a <newline> plus one or more blank lines, leading or trailing blank lines shall not result in empty records at the beginning or end of the input, and a <newline> shall always be a field separator, no matter what the value of **FS** is.
RSTART	The starting position of the string matched by the **match** function, numbering from 1. This shall always be equivalent to the return value of the **match** function.
SUBSEP	The subscript separator string for multi-dimensional arrays; the default value is implementation-defined.

Regular Expressions

The *awk* utility shall make use of the extended regular expression notation (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.4, Extended Regular Expressions) except that it shall allow the use of C-language conventions for escaping special characters within the EREs, as specified in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`′\\′`, `′\a′`, `′\b′`, `′\f′`, `′\n′`, `′\r′`, `′\t′`, `′\v′`) and the following table; these escape sequences shall be recognized both inside and outside bracket expressions. Note that records need not be separated by <newline>s and string constants can contain <newline>s, so even the `"\n"` sequence is valid in *awk* EREs. Using a slash character within an ERE requires the escaping shown in the following table.

Table 4-2 Escape Sequences in *awk*

Escape Sequence	Description	Meaning
`\"`	Backslash quotation-mark	Quotation-mark character
`\/`	Backslash slash	Slash character
`\ddd`	A backslash character followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0 (that is, representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the one, two, or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-defined. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading `'\'` for each byte.
`\c`	A backslash character followed by any character not described in this table or in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`)	Undefined

A regular expression can be matched against a specific field or string by using one of the two regular expression matching operators, `'~'` and `"!~"`. These operators shall interpret their right-hand operand as a regular expression and their left-hand operand as a string. If the regular expression matches the string, the `'~'` expression shall evaluate to a value of 1, and the `"!~"` expression shall evaluate to a value of 0. (The regular expression matching operation is as defined by the term matched in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.1, Regular Expression Definitions, where a match occurs on any part of the string unless the regular expression is limited with the circumflex or dollar sign special characters.) If the regular expression does not match the string, the `'~'` expression shall evaluate to a value of 0, and the `"!~"` expression shall evaluate to a value of 1. If the right-hand operand is any expression other than the lexical token **ERE**, the string value of the expression shall be interpreted as an extended regular expression, including the escape conventions described above. Note that these same escape conventions shall also be applied in determining the value of a string literal (the lexical token **STRING**), and thus shall be applied a second time when a string literal is used in this context.

When an **ERE** token appears as an expression in any context other than as the right-hand of the `'~'` or `"!~"` operator or as one of the built-in function arguments described below, the value of the resulting expression shall be the equivalent of:

```
$0 ~ /ere/
```

The *ere* argument to the **gsub**, **match**, **sub** functions, and the *fs* argument to the **split** function (see **String Functions** (on page 2369)) shall be interpreted as extended regular expressions. These can be either **ERE** tokens or arbitrary expressions, and shall be interpreted in the same manner as the right-hand side of the `'~'` or `"!~"` operator.

An extended regular expression can be used to separate fields by using the **–F** *ERE* option or by assigning a string containing the expression to the built-in variable **FS**. The default value of the **FS** variable shall be a single <space>. The following describes **FS** behavior:

1. If **FS** is a null string, the behavior is unspecified.
2. If **FS** is a single character:
    a. If **FS** is <space>, skip leading and trailing <blank>s; fields shall be delimited by sets of one or more <blank>s.
    b. Otherwise, if **FS** is any other character *c*, fields shall be delimited by each single occurrence of *c*.
3. Otherwise, the string value of **FS** shall be considered to be an extended regular expression. Each occurrence of a sequence matching the extended regular expression shall delimit fields.

Except for the `'~'` and `"!~"` operators, and in the **gsub**, **match**, **split**, and **sub** built-in functions, ERE matching shall be based on input records; that is, record separator characters (the first character of the value of the variable **RS**, <newline> by default) cannot be embedded in the expression, and no expression shall match the record separator character. If the record separator is not <newline>, <newline>s embedded in the expression can be matched. For the `'~'` and `"!~"` operators, and in those four built-in functions, ERE matching shall be based on text strings; that is, any character (including <newline> and the record separator) can be embedded in the pattern, and an appropriate pattern shall match any character. However, in all *awk* ERE matching, the use of one or more NUL characters in the pattern, input record, or text string produces undefined results.

Patterns

A *pattern* is any valid *expression*, a range specified by two expressions separated by comma, or one of the two special patterns **BEGIN** or **END**.

Special Patterns

The *awk* utility shall recognize two special patterns, **BEGIN** and **END**. Each **BEGIN** pattern shall be matched once and its associated action executed before the first record of input is read (except possibly by use of the **getline** function—see **Input/Output and General Functions** (on page 2370)—in a prior **BEGIN** action) and before command line assignment is done. Each **END** pattern shall be matched once and its associated action executed after the last record of input has been read. These two patterns shall have associated actions.

BEGIN and **END** shall not combine with other patterns. Multiple **BEGIN** and **END** patterns shall be allowed. The actions associated with the **BEGIN** patterns shall be executed in the order specified in the program, as are the **END** actions. An **END** pattern can precede a **BEGIN** pattern in a program.

If an *awk* program consists of only actions with the pattern **BEGIN**, and the **BEGIN** action contains no **getline** function, *awk* shall exit without reading its input when the last statement in the last **BEGIN** action is executed. If an *awk* program consists of only actions with the pattern **END** or only actions with the patterns **BEGIN** and **END**, the input shall be read before the statements in the **END** actions are executed.

Expression Patterns

An expression pattern shall be evaluated as if it were an expression in a Boolean context. If the result is true, the pattern shall be considered to match, and the associated action (if any) shall be executed. If the result is false, the action shall not be executed.

Pattern Ranges

A pattern range consists of two expressions separated by a comma; in this case, the action shall be performed for all records between a match of the first expression and the following match of the second expression, inclusive. At this point, the pattern range can be repeated starting at input records subsequent to the end of the matched range.

Actions

An action is a sequence of statements as shown in the grammar in **Grammar** (on page 2372). Any single statement can be replaced by a statement list enclosed in braces. The application shall ensure that statements in a statement list are separated by <newline>s or semicolons. Statements in a statement list shall be executed sequentially in the order that they appear.

The *expression* acting as the conditional in an **if** statement shall be evaluated and if it is non-zero or non-null, the following *statement* shall be executed; otherwise, if **else** is present, the statement following the **else** shall be executed.

The **if**, **while**, **do**...**while**, **for**, **break**, and **continue** statements are based on the ISO C standard (see Section 1.7.2 (on page 2207)), except that the Boolean expressions shall be treated as described in **Expressions in awk** (on page 2358), and except in the case of:

```
for (variable in array)
```

which shall iterate, assigning each *index* of *array* to *variable* in an unspecified order. The results of adding new elements to *array* within such a **for** loop are undefined. If a **break** or **continue** statement occurs outside of a loop, the behavior is undefined.

The **delete** statement shall remove an individual array element. Thus, the following code deletes an entire array:

```
for (index in array)
    delete array[index]
```

The **next** statement shall cause all further processing of the current input record to be abandoned. The behavior is undefined if a **next** statement appears or is invoked in a **BEGIN** or **END** action.

The **exit** statement shall invoke all **END** actions in the order in which they occur in the program source and then terminate the program without reading further input. An **exit** statement inside an **END** action shall terminate the program without further execution of **END** actions. If an expression is specified in an **exit** statement, its numeric value shall be the exit status of *awk*, unless subsequent errors are encountered or a subsequent **exit** statement with an expression is executed.

Output Statements

Both **print** and **printf** statements shall write to standard output by default. The output shall be written to the location specified by *output_redirection* if one is supplied, as follows:

```
> expression
>> expression
| expression
```

In all cases, the *expression* shall be evaluated to produce a string that is used as a pathname into which to write (for `'>'` or `">>"`) or as a command to be executed (for `'|'`). Using the first two forms, if the file of that name is not currently open, it shall be opened, creating it if necessary and using the first form, truncating the file. The output then shall be appended to the file. As long as the file remains open, subsequent calls in which *expression* evaluates to the same string value shall simply append output to the file. The file remains open until the **close** function (see **Input/Output and General Functions** (on page 2370)) is called with an expression that evaluates to the same string value.

The third form shall write output onto a stream piped to the input of a command. The stream shall be created if no stream is currently open with the value of *expression* as its command name. The stream created shall be equivalent to one created by a call to the *popen*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x with the value of *expression* as the *command* argument and a value of *w* as the *mode* argument. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall write output to the existing stream. The stream shall remain open until the **close** function (see **Input/Output and General Functions** (on page 2370)) is called with an expression that evaluates to the same string value. At that time, the stream shall be closed as if by a call to the *pclose*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x.

As described in detail by the grammar in **Grammar** (on page 2372), these output statements shall take a comma-separated list of *expression*s referred to in the grammar by the non-terminal symbols **expr_list**, **print_expr_list**, or **print_expr_list_opt**. This list is referred to here as the *expression list*, and each member is referred to as an *expression argument*.

The **print** statement shall write the value of each expression argument onto the indicated output stream separated by the current output field separator (see variable **OFS** above), and terminated by the output record separator (see variable **ORS** above). All expression arguments shall be taken as strings, being converted if necessary; this conversion shall be as described in **Expressions in awk** (on page 2358), with the exception that the **printf** format in **OFMT** shall be used instead of the value in **CONVFMT**. An empty expression list shall stand for the whole input record ($0).

The **printf** statement shall produce output based on a notation similar to the File Format Notation used to describe file formats in this volume of IEEE Std 1003.1-200x (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation). Output shall be produced as specified with the first expression argument as the string *format* and subsequent expression arguments as the strings *arg1* to *argn*, inclusive, with the following exceptions:

1. The *format* shall be an actual character string rather than a graphical representation. Therefore, it cannot contain empty character positions. The <space> in the *format* string, in any context other than a *flag* of a conversion specification, shall be treated as an ordinary character that is copied to the output.

2. If the character set contains a `'Δ'` character and that character appears in the *format* string, it shall be treated as an ordinary character that is copied to the output.

3. The *escape sequences* beginning with a backslash character shall be treated as sequences of ordinary characters that are copied to the output. Note that these same sequences shall be interpreted lexically by *awk* when they appear in literal strings, but they shall not be treated specially by the **printf** statement.
4. A *field width* or *precision* can be specified as the `'*'` character instead of a digit string. In this case the next argument from the expression list shall be fetched and its numeric value taken as the field width or precision.
5. The implementation shall not precede or follow output from the *d* or *u* conversion specifications with <blank>s not specified by the *format* string.
6. The implementation shall not precede output from the *o* conversion specification with leading zeros not specified by the *format* string.
7. For the *c* conversion specification: if the argument has a numeric value, the character whose encoding is that value shall be output. If the value is zero or is not the encoding of any character in the character set, the behavior is undefined. If the argument does not have a numeric value, the first character of the string value shall be output; if the string does not contain any characters, the behavior is undefined.
8. For each conversion specification that consumes an argument, the next expression argument shall be evaluated. With the exception of the *c* conversion, the value shall be converted (according to the rules specified in **Expressions in awk** (on page 2358)) to the appropriate type for the conversion specification.
9. If there are insufficient expression arguments to satisfy all the conversion specifications in the *format* string, the behavior is undefined.
10. If any character sequence in the *format* string begins with a `'%'` character, but does not form a valid conversion specification, the behavior is unspecified.

Both **print** and **printf** can output at least {LINE_MAX} bytes.

Functions

The *awk* language has a variety of built-in functions: arithmetic, string, input/output, and general.

Arithmetic Functions

The arithmetic functions, except for **int**, shall be based on the ISO C standard (see Section 1.7.2 (on page 2207)). The behavior is undefined in cases where the ISO C standard specifies that an error be returned or that the behavior is undefined. Although the grammar (see **Grammar** (on page 2372)) permits built-in functions to appear with no arguments or parentheses, unless the argument or parentheses are indicated as optional in the following list (by displaying them within the `"[]"` brackets), such use is undefined.

atan2(*y*,*x*)	Return arctangent of y/x in radians in the range $[-\pi,\pi]$.
cos(*x*)	Return cosine of *x*, where *x* is in radians.
sin(*x*)	Return sine of *x*, where *x* is in radians.
exp(*x*)	Return the exponential function of *x*.
log(*x*)	Return the natural logarithm of *x*.
sqrt(*x*)	Return the square root of *x*.

int(*x*)	Return the argument truncated to an integer. Truncation shall be toward 0 when $x > 0$.
rand( )	Return a random number *n*, such that $0 \le n < 1$.
srand(**[***expr***]**)	Set the seed value for *rand* to *expr* or use the time of day if *expr* is omitted. The previous seed value shall be returned.

String Functions

The string functions in the following list shall be supported. Although the grammar (see **Grammar** (on page 2372)) permits built-in functions to appear with no arguments or parentheses, unless the argument or parentheses are indicated as optional in the following list (by displaying them within the "[ ]" brackets), such use is undefined.

gsub(*ere*, *repl***[**, *in***]**)	Behave like **sub** (see below), except that it shall replace all occurrences of the regular expression (like the *ed* utility global substitute) in $0 or in the *in* argument, when specified.
index(*s*, *t*)	Return the position, in characters, numbering from 1, in string *s* where string *t* first occurs, or zero if it does not occur at all.
length[(**[***s***]**)**]**	Return the length, in characters, of its argument taken as a string, or of the whole record, $0, if there is no argument.
match(*s*, *ere*)	Return the position, in characters, numbering from 1, in string *s* where the extended regular expression *ere* occurs, or zero if it does not occur at all. RSTART shall be set to the starting position (which is the same as the returned value), zero if no match is found; RLENGTH shall be set to the length of the matched string, –1 if no match is found.
split(*s*, *a***[**, *fs* **]**)	Split the string *s* into array elements *a*[1], *a*[2], . . ., *a*[*n*], and return *n*. All elements of the array shall be deleted before the split is performed. The separation shall be done with the ERE *fs* or with the field separator **FS** if *fs* is not given. Each array element shall have a string value when created and, if appropriate, the array element shall be considered a numeric string (see **Expressions in awk** (on page 2358)). The effect of a null string as the value of *fs* is unspecified.
sprintf(*fmt*, *expr*, *expr*, . . .)	Format the expressions according to the **printf** format given by *fmt* and return the resulting string.
sub(*ere*, *repl***[**, *in* **]**)	Substitute the string *repl* in place of the first instance of the extended regular expression *ERE* in string *in* and return the number of substitutions. An ampersand ('&') appearing in the string *repl* shall be replaced by the string from *in* that matches the ERE. An ampersand preceded with a backslash ('\') shall be interpreted as the literal ampersand character. An occurrence of two consecutive backslashes shall be interpreted as just a single literal backslash character. Any other occurrence of a backslash (for example, preceding any other character) shall be treated as a literal backslash character. Note that if *repl* is a string literal (the lexical token **STRING**; see **Grammar** (on page 2372)), the handling of the ampersand character occurs after any lexical processing, including any lexical backslash escape sequence processing. If *in* is specified and it is not an lvalue (see **Expressions in awk** (on page 2358)), the behavior is undefined. If *in* is omitted, *awk*

shall use the current record ($0) in its place.

substr(*s*, *m***[**, *n* **]**)
Return the at most *n*-character substring of *s* that begins at position *m*, numbering from 1. If *n* is omitted, or if *n* specifies more characters than are left in the string, the length of the substring shall be limited by the length of the string *s*.

tolower(*s*) Return a string based on the string *s*. Each character in *s* that is an uppercase letter specified to have a **tolower** mapping by the *LC_CTYPE* category of the current locale shall be replaced in the returned string by the lowercase letter specified by the mapping. Other characters in *s* shall be unchanged in the returned string.

toupper(*s*) Return a string based on the string *s*. Each character in *s* that is a lowercase letter specified to have a **toupper** mapping by the *LC_CTYPE* category of the current locale is replaced in the returned string by the uppercase letter specified by the mapping. Other characters in *s* are unchanged in the returned string.

All of the preceding functions that take *ERE* as a parameter expect a pattern or a string valued expression that is a regular expression as defined in **Regular Expressions** (on page 2363).

Input/Output and General Functions

The input/output and general functions are:

close(*expression*)
Close the file or pipe opened by a **print** or **printf** statement or a call to **getline** with the same string-valued *expression*. The limit on the number of open *expression* arguments is implementation-defined. If the close was successful, the function shall return zero; otherwise, it shall return non-zero.

expression | **getline [***var***]**
Read a record of input from a stream piped from the output of a command. The stream shall be created if no stream is currently open with the value of *expression* as its command name. The stream created shall be equivalent to one created by a call to the *popen*( ) function with the value of *expression* as the *command* argument and a value of *r* as the *mode* argument. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall read subsequent records from the stream. The stream shall remain open until the **close** function is called with an expression that evaluates to the same string value. At that time, the stream shall be closed as if by a call to the *pclose*( ) function. If *var* is omitted, $0 and **NF** shall be set; otherwise, *var* shall be set and, if appropriate, it shall be considered a numeric string (see **Expressions in awk** (on page 2358)).

The **getline** operator can form ambiguous constructs when there are unparenthesized operators (including concatenate) to the left of the `'|'` (to the beginning of the expression containing **getline**). In the context of the `'$'` operator, `'|'` shall behave as if it had a lower precedence than `'$'`. The result of evaluating other operators is unspecified, and conforming applications shall parenthesize properly all such usages.

getline Set $0 to the next input record from the current input file. This form of **getline** shall set the **NF**, **NR**, and **FNR** variables.

getline *var* Set variable *var* to the next input record from the current input file and, if appropriate, *var* shall be considered a numeric string (see **Expressions in awk** (on page 2358)). This form of **getline** shall set the **FNR** and **NR** variables.

getline [*var***]** < *expression*

Read the next record of input from a named file. The *expression* shall be evaluated to produce a string that is used as a pathname. If the file of that name is not currently open, it shall be opened. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall read subsequent records from the file. The file shall remain open until the **close** function is called with an expression that evaluates to the same string value. If *var* is omitted, $0 and **NF** shall be set; otherwise, *var* shall be set and, if appropriate, it shall be considered a numeric string (see **Expressions in awk** (on page 2358)).

The **getline** operator can form ambiguous constructs when there are unparenthesized binary operators (including concatenate) to the right of the '<' (up to the end of the expression containing the **getline**). The result of evaluating such a construct is unspecified, and conforming applications shall parenthesize properly all such usages.

system(*expression*)

Execute the command given by *expression* in a manner equivalent to the *system*() function defined in the System Interfaces volume of IEEE Std 1003.1-200x and return the exit status of the command.

All forms of **getline** shall return 1 for successful input, zero for end-of-file, and −1 for an error.

Where strings are used as the name of a file or pipeline, the application shall ensure that the strings are textually identical. The terminology ‘‘same string value’’ implies that ‘‘equivalent strings’’, even those that differ only by <space>s, represent different files.

User-Defined Functions

The *awk* language also provides user-defined functions. Such functions can be defined as:

```
function name([parameter, ...]) { statements }
```

A function can be referred to anywhere in an *awk* program; in particular, its use can precede its definition. The scope of a function is global.

Function parameters, if present, can be either scalars or arrays; the behavior is undefined if an array name is passed as a parameter that the function uses as a scalar, or if a scalar expression is passed as a parameter that the function uses as an array. Function parameters shall be passed by value if scalar and by reference if array name.

The number of parameters in the function definition need not match the number of parameters in the function call. Excess formal parameters can be used as local variables. If fewer arguments are supplied in a function call than are in the function definition, the extra parameters that are used in the function body as scalars shall evaluate to the uninitialized value until they are otherwise initialized, and the extra parameters that are used in the function body as arrays shall be treated as uninitialized arrays where each element evaluates to the uninitialized value until otherwise initialized.

When invoking a function, no white space can be placed between the function name and the opening parenthesis. Function calls can be nested and recursive calls can be made upon functions. Upon return from any nested or recursive function call, the values of all of the calling function's parameters shall be unchanged, except for array parameters passed by reference. The **return** statement can be used to return a value. If a **return** statement appears outside of a function definition, the behavior is undefined.

In the function definition, <newline>s shall be optional before the opening brace and after the closing brace. Function definitions can appear anywhere in the program where a *pattern-action* pair is allowed.

Grammar

The grammar in this section and the lexical conventions in the following section shall together describe the syntax for *awk* programs. The general conventions for this style of grammar are described in Section 1.10 (on page 2220). A valid program can be represented as the non-terminal symbol *program* in the grammar. This formal syntax shall take precedence over the preceding text syntax description.

```
%token NAME NUMBER STRING ERE
%token FUNC_NAME   /* Name followed by '(' without white space. */

/* Keywords  */
%token       Begin   End
/*          'BEGIN' 'END'                            */

%token       Break   Continue   Delete   Do   Else
/*          'break' 'continue' 'delete' 'do' 'else'  */

%token       Exit   For   Function   If   In
/*          'exit' 'for' 'function' 'if' 'in'        */

%token       Next   Print   Printf   Return   While
/*          'next' 'print' 'printf' 'return' 'while' */

/* Reserved function names */
%token BUILTIN_FUNC_NAME
            /* One token for the following:
             * atan2 cos sin exp log sqrt int rand srand
             * gsub index length match split sprintf sub
             * substr tolower toupper close system
             */
%token GETLINE
            /* Syntactically different from other built-ins. */

/* Two-character tokens. */
%token ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN POW_ASSIGN
/*     '+='       '-='       '*='       '/='       '%='       '^=' */

%token OR   AND  NO_MATCH   EQ   LE   GE   NE   INCR  DECR  APPEND
/*     '||' '&&' '!~' '==' '<=' '>=' '!=' '++'  '--'  '>>'   */

/* One-character tokens. */
%token '{' '}' '(' ')' '[' ']' ',' ';' NEWLINE
%token '+' '-' '*' '%' '^' '!' '>' '<' '|' '?' ':' '~' '$' '='

%start program
%%

program          : item_list
                 | actionless_item_list
                 ;

item_list        : newline_opt
                 | actionless_item_list item terminator
                 | item_list            item terminator
                 | item_list          action terminator
                 ;
```

```
actionless_item_list : item_list            pattern terminator
                 | actionless_item_list pattern terminator
                 ;

item             : pattern action
                 | Function NAME      '(' param_list_opt ')'
                       newline_opt action
                 | Function FUNC_NAME '(' param_list_opt ')'
                       newline_opt action
                 ;

param_list_opt   : /* empty */
                 | param_list
                 ;

param_list       : NAME
                 | param_list ',' NAME
                 ;

pattern          : Begin
                 | End
                 | expr
                 | expr ',' newline_opt expr
                 ;

action           : '{' newline_opt                             '}'
                 | '{' newline_opt terminated_statement_list   '}'
                 | '{' newline_opt unterminated_statement_list '}'
                 ;

terminator       : terminator ';'
                 | terminator NEWLINE
                 |            ';'
                 |            NEWLINE
                 ;

terminated_statement_list : terminated_statement
                 | terminated_statement_list terminated_statement
                 ;

unterminated_statement_list : unterminated_statement
                 | terminated_statement_list unterminated_statement
                 ;

terminated_statement : action newline_opt
                 | If '(' expr ')' newline_opt terminated_statement
                 | If '(' expr ')' newline_opt terminated_statement
                       Else newline_opt terminated_statement
                 | While '(' expr ')' newline_opt terminated_statement
                 | For '(' simple_statement_opt ';'
                      expr_opt ';' simple_statement_opt ')' newline_opt
                      terminated_statement
                 | For '(' NAME In NAME ')' newline_opt
                      terminated_statement
                 | ';' newline_opt
                 | terminatable_statement NEWLINE newline_opt
                 | terminatable_statement ';'     newline_opt
```

```
                 ;
unterminated_statement : terminatable_statement
                 | If ’(’ expr ’)’ newline_opt unterminated_statement
                 | If ’(’ expr ’)’ newline_opt terminated_statement
                      Else newline_opt unterminated_statement
                 | While ’(’ expr ’)’ newline_opt unterminated_statement
                 | For ’(’ simple_statement_opt ’;’
                  expr_opt ’;’ simple_statement_opt ’)’ newline_opt
                      unterminated_statement
                 | For ’(’ NAME In NAME ’)’ newline_opt
                      unterminated_statement
                 ;
terminatable_statement : simple_statement
                 | Break
                 | Continue
                 | Next
                 | Exit expr_opt
                 | Return expr_opt
                 | Do newline_opt terminated_statement While ’(’ expr ’)’
                 ;
simple_statement_opt : /* empty */
                 | simple_statement
                 ;
simple_statement : Delete NAME ’[’ expr_list ’]’
                 | expr
                 | print_statement
                 ;
print_statement  : simple_print_statement
                 | simple_print_statement output_redirection
                 ;
simple_print_statement : Print  print_expr_list_opt
                 | Print  ’(’ multiple_expr_list ’)’
                 | Printf print_expr_list
                 | Printf ’(’ multiple_expr_list ’)’
                 ;
output_redirection : ’>’    expr
                 | APPEND expr
                 | ’|’    expr
                 ;
expr_list_opt    : /* empty */
                 | expr_list
                 ;
expr_list        : expr
                 | multiple_expr_list
                 ;
multiple_expr_list : expr ’,’ newline_opt expr
                 | multiple_expr_list ’,’ newline_opt expr
```

```
                  ;

expr_opt          : /* empty */
                  | expr
                  ;

expr              : unary_expr
                  | non_unary_expr
                  ;

unary_expr        : '+' expr
                  | '-' expr
                  | unary_expr '^'      expr
                  | unary_expr '*'      expr
                  | unary_expr '/'      expr
                  | unary_expr '%'      expr
                  | unary_expr '+'      expr
                  | unary_expr '-'      expr
                  | unary_expr          non_unary_expr
                  | unary_expr '<'      expr
                  | unary_expr LE       expr
                  | unary_expr NE       expr
                  | unary_expr EQ       expr
                  | unary_expr '>'      expr
                  | unary_expr GE       expr
                  | unary_expr '~'      expr
                  | unary_expr NO_MATCH expr
                  | unary_expr In NAME
                  | unary_expr AND newline_opt expr
                  | unary_expr OR  newline_opt expr
                  | unary_expr '?' expr ':' expr
                  | unary_input_function
                  ;

non_unary_expr    : '(' expr ')'
                  | '!' expr
                  | non_unary_expr '^'      expr
                  | non_unary_expr '*'      expr
                  | non_unary_expr '/'      expr
                  | non_unary_expr '%'      expr
                  | non_unary_expr '+'      expr
                  | non_unary_expr '-'      expr
                  | non_unary_expr          non_unary_expr
                  | non_unary_expr '<'      expr
                  | non_unary_expr LE       expr
                  | non_unary_expr NE       expr
                  | non_unary_expr EQ       expr
                  | non_unary_expr '>'      expr
                  | non_unary_expr GE       expr
                  | non_unary_expr '~'      expr
                  | non_unary_expr NO_MATCH expr
                  | non_unary_expr In NAME
                  | '(' multiple_expr_list ')' In NAME
                  | non_unary_expr AND newline_opt expr
```

```
                 | non_unary_expr OR  newline_opt expr
                 | non_unary_expr '?' expr ':' expr
                 | NUMBER
                 | STRING
                 | lvalue
                 | ERE
                 | lvalue INCR
                 | lvalue DECR
                 | INCR lvalue
                 | DECR lvalue
                 | lvalue POW_ASSIGN expr
                 | lvalue MOD_ASSIGN expr
                 | lvalue MUL_ASSIGN expr
                 | lvalue DIV_ASSIGN expr
                 | lvalue ADD_ASSIGN expr
                 | lvalue SUB_ASSIGN expr
                 | lvalue '=' expr
                 | FUNC_NAME '(' expr_list_opt ')'
                      /* no white space allowed before '(' */
                 | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
                 | BUILTIN_FUNC_NAME
                 | non_unary_input_function
                 ;

print_expr_list_opt : /* empty */
                 | print_expr_list
                 ;

print_expr_list  : print_expr
                 | print_expr_list ',' newline_opt print_expr
                 ;

print_expr       : unary_print_expr
                 | non_unary_print_expr
                 ;

unary_print_expr : '+' print_expr
                 | '-' print_expr
                 | unary_print_expr '^'      print_expr
                 | unary_print_expr '*'      print_expr
                 | unary_print_expr '/'      print_expr
                 | unary_print_expr '%'      print_expr
                 | unary_print_expr '+'      print_expr
                 | unary_print_expr '-'      print_expr
                 | unary_print_expr          non_unary_print_expr
                 | unary_print_expr '~'      print_expr
                 | unary_print_expr NO_MATCH print_expr
                 | unary_print_expr In NAME
                 | unary_print_expr AND newline_opt print_expr
                 | unary_print_expr OR  newline_opt print_expr
                 | unary_print_expr '?' print_expr ':' print_expr
                 ;

non_unary_print_expr : '(' expr ')'
                 | '!' print_expr
```

```
                      | non_unary_print_expr '^'      print_expr
                      | non_unary_print_expr '*'      print_expr
                      | non_unary_print_expr '/'      print_expr
                      | non_unary_print_expr '%'      print_expr
                      | non_unary_print_expr '+'      print_expr
                      | non_unary_print_expr '-'      print_expr
                      | non_unary_print_expr          non_unary_print_expr
                      | non_unary_print_expr '~'      print_expr
                      | non_unary_print_expr NO_MATCH print_expr
                      | non_unary_print_expr In NAME
                      | '(' multiple_expr_list ')' In NAME
                      | non_unary_print_expr AND newline_opt print_expr
                      | non_unary_print_expr OR  newline_opt print_expr
                      | non_unary_print_expr '?' print_expr ':' print_expr
                      | NUMBER
                      | STRING
                      | lvalue
                      | ERE
                      | lvalue INCR
                      | lvalue DECR
                      | INCR lvalue
                      | DECR lvalue
                      | lvalue POW_ASSIGN print_expr
                      | lvalue MOD_ASSIGN print_expr
                      | lvalue MUL_ASSIGN print_expr
                      | lvalue DIV_ASSIGN print_expr
                      | lvalue ADD_ASSIGN print_expr
                      | lvalue SUB_ASSIGN print_expr
                      | lvalue '=' print_expr
                      | FUNC_NAME '(' expr_list_opt ')'
                          /* no white space allowed before '(' */
                      | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
                      | BUILTIN_FUNC_NAME
                      ;

    lvalue            : NAME
                      | NAME '[' expr_list ']'
                      | '$' expr
                      ;

    non_unary_input_function : simple_get
                      | simple_get '<' expr
                      | non_unary_expr '|' simple_get
                      ;

    unary_input_function : unary_expr '|' simple_get
                      ;

    simple_get        : GETLINE
                      | GETLINE lvalue
                      ;

    newline_opt       : /* empty */
                      | newline_opt NEWLINE
                      ;
```

This grammar has several ambiguities that shall be resolved as follows:

- Operator precedence and associativity shall be as described in Table 4-1 (on page 2358).
- In case of ambiguity, an **else** shall be associated with the most immediately preceding **if** that would satisfy the grammar.
- In some contexts, a slash (`'/'`) that is used to surround an ERE could also be the division operator. This shall be resolved in such a way that wherever the division operator could appear, a slash is assumed to be the division operator. (There is no unary division operator.)

One convention that might not be obvious from the formal grammar is where <newline>s are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape <newline>s between any lexical tokens. In addition, <newline>s without backslashes can follow a comma, an open brace, logical AND operator (`"&&"`), logical OR operator (`"||"`), the **do** keyword, the **else** keyword, and the closing parenthesis of an **if**, **for**, or **while** statement. For example:

```
{ print  $1,
         $2 }
```

Lexical Conventions

The lexical conventions for *awk* programs, with respect to the preceding grammar, shall be as follows:

1. Except as noted, *awk* shall recognize the longest possible token or delimiter beginning at a given point.
2. A comment shall consist of any characters beginning with the number sign character and terminated by, but excluding the next occurrence of, a <newline>. Comments shall have no effect, except to delimit lexical tokens.
3. The <newline> shall be recognized as the token **NEWLINE**.
4. A backslash character immediately followed by a <newline> shall have no effect.
5. The token **STRING** shall represent a string constant. A string constant shall begin with the character `'"'`. Within a string constant, a backslash character shall be considered to begin an escape sequence as specified in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`). In addition, the escape sequences in Table 4-2 (on page 2364) shall be recognized. A <newline> shall not occur within a string constant. A string constant shall be terminated by the first unescaped occurrence of the character `'"'` after the one that begins the string constant. The value of the string shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting `'"'` characters.
6. The token **ERE** represents an extended regular expression constant. An ERE constant shall begin with the slash character. Within an ERE constant, a backslash character shall be considered to begin an escape sequence as specified in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation. In addition, the escape sequences in Table 4-2 (on page 2364) shall be recognized. The application shall ensure that a <newline> does not occur within an ERE constant. An ERE constant shall be terminated by the first unescaped occurrence of the slash character after the one that begins the ERE constant. The extended regular expression represented by the ERE constant shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting slash characters.

7. A <blank> shall have no effect, except to delimit lexical tokens or within **STRING** or **ERE** tokens.

8. The token **NUMBER** shall represent a numeric constant. Its form and numeric value shall be equivalent to either of the tokens **floating-constant** or **integer-constant** as specified by the ISO C standard, with the following exceptions:

   a. An integer constant cannot begin with 0x or include the hexadecimal digits `'a'`, `'b'`, `'c'`, `'d'`, `'e'`, `'f'`, `'A'`, `'B'`, `'C'`, `'D'`, `'E'`, or `'F'`.

   b. The value of an integer constant beginning with 0 shall be taken in decimal rather than octal.

   c. An integer constant cannot include a suffix (`'u'`, `'U'`, `'l'`, or `'L'`).

   d. A floating constant cannot include a suffix (`'f'`, `'F'`, `'l'`, or `'L'`).

   If the value is too large or too small to be representable (see Section 1.7.2 (on page 2207)), the behavior is undefined.

9. A sequence of underscores, digits, and alphabetics from the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set), beginning with an underscore or alphabetic, shall be considered a word.

10. The following words are keywords that shall be recognized as individual tokens; the name of the token is the same as the keyword:

    BEGIN
    break
    continue
    delete
    do
    else
    END
    exit
    for
    function
    getline
    if
    in
    next
    print
    printf
    return
    while

11. The following words are names of built-in functions and shall be recognized as the token **BUILTIN_FUNC_NAME**:

    atan2
    close
    cos
    exp
    gsub
    index
    int
    length
    log
    match
    rand
    sin
    split
    sprintf
    sqrt
    srand
    sub
    substr
    system
    tolower
    toupper

    The above-listed keywords and names of built-in functions are considered reserved words.

12. The token **NAME** shall consist of a word that is not a keyword or a name of a built-in function and is not followed immediately (without any delimiters) by the `'('` character.

13. The token **FUNC_NAME** shall consist of a word that is not a keyword or a name of a built-in function, followed immediately (without any delimiters) by the `'('` character. The `'('` character shall not be included as part of the token.

14. The following two-character sequences shall be recognized as the named tokens:

Token Name	Sequence	Token Name	Sequence
ADD_ASSIGN	+=	**NO_MATCH**	!~
SUB_ASSIGN	-=	**EQ**	==
MUL_ASSIGN	*=	**LE**	<=
DIV_ASSIGN	/=	**GE**	>=
MOD_ASSIGN	%=	**NE**	!=
POW_ASSIGN	^=	**INCR**	++
OR	||	**DECR**	--
AND	&&	**APPEND**	>>

15. The following single characters shall be recognized as tokens whose names are the character:

    ```
    <newline> { } ( ) [ ] , ; + – * % ˆ ! > < | ? : ˜ $ =
    ```

There is a lexical ambiguity between the token **ERE** and the tokens `'/'` and **DIV_ASSIGN**. When an input sequence begins with a slash character in any syntactic context where the token `'/'` or **DIV_ASSIGN** could appear as the next token in a valid program, the longer of those two tokens that can be recognized shall be recognized. In any other syntactic context where the token **ERE** could appear as the next token in a valid program, the token **ERE** shall be recognized.

EXIT STATUS

The following exit values shall be returned:

0	All input files were processed successfully.
>0	An error occurred.

The exit status can be altered within the program by using an **exit** expression.

CONSEQUENCES OF ERRORS

If any *file* operand is specified and the named file cannot be accessed, *awk* shall write a diagnostic message to standard error and terminate without any further action.

If the program specified by either the *program* operand or a *progfile* operand is not a valid *awk* program (as specified in the EXTENDED DESCRIPTION section), the behavior is undefined.

APPLICATION USAGE

The **index**, **length**, **match**, and **substr** functions should not be confused with similar functions in the ISO C standard; the *awk* versions deal with characters, while the ISO C standard deals with bytes.

Because the concatenation operation is represented by adjacent expressions rather than an explicit operator, it is often necessary to use parentheses to enforce the proper evaluation precedence.

EXAMPLES

The *awk* program specified in the command line is most easily specified within single-quotes (for example, '*program*') for applications using *sh*, because *awk* programs commonly contain characters that are special to the shell, including double-quotes. In the cases where an *awk* program contains single-quote characters, it is usually easiest to specify most of the program as strings within single-quotes concatenated by the shell with quoted single-quote characters. For example:

```
awk '/'\''/ { print "quote:", $0 }'
```

prints all lines from the standard input containing a single-quote character, prefixed with *quote*:.

The following are examples of simple *awk* programs:

1. Write to the standard output all input lines for which field 3 is greater than 5:

   ```
   $3 > 5
   ```

2. Write every tenth line:

   ```
   (NR % 10) == 0
   ```

3. Write any line with a substring matching the regular expression:

   ```
   /(G|D)(2[0–9][[:alpha:]]*)/
   ```

4. Print any line with a substring containing a `'G'` or `'D'`, followed by a sequence of digits and characters. This example uses character classes **digit** and **alpha** to match language-independent digit and alphabetic characters respectively:

   ```
   /(G|D)([[:digit:][:alpha:]]*)/
   ```

5. Write any line in which the second field matches the regular expression and the fourth field does not:

   ```
   $2 ~ /xyz/ && $4 !~ /xyz/
   ```

6. Write any line in which the second field contains a backslash:

   ```
   $2 ~ /\\/
   ```

7. Write any line in which the second field contains a backslash. Note that backslash escapes are interpreted twice, once in lexical processing of the string and once in processing the regular expression:

   ```
   $2 ~ "\\\\"
   ```

8. Write the second to the last and the last field in each line. Separate the fields by a colon:

   ```
   {OFS=":";print $(NF-1), $NF}
   ```

9. Write the line number and number of fields in each line. The three strings representing the line number, the colon, and the number of fields are concatenated and that string is written to standard output:

   ```
   {print NR ":" NF}
   ```

10. Write lines longer than 72 characters:

    ```
    length($0) > 72
    ```

11. Write first two fields in opposite order separated by the **OFS**:

    ```
    { print $2, $1 }
    ```

12. Same, with input fields separated by comma or <space>s and <tab>s, or both:

    ```
    BEGIN { FS = ",[ \t]*|[ \t]+" }
          { print $2, $1 }
    ```

13. Add up first column, print sum, and average:

    ```
          {s += $1 }
    END   {print "sum is ", s, " average is", s/NR}
    ```

14. Write fields in reverse order, one per line (many lines out for each line in):

    ```
    { for (i = NF; i > 0; --i) print $i }
    ```

15. Write all lines between occurrences of the strings **start** and **stop**:

    ```
    /start/, /stop/
    ```

16. Write all lines whose first field is different from the previous one:

    ```
    $1 != prev { print; prev = $1 }
    ```

17. Simulate *echo*:

    ```
    BEGIN  {
            for (i = 1; i < ARGC; ++i)
            printf("%s%s", ARGV[i], i==ARGC-1?"\n":" ")
    ```

```
        }
```

18. Write the path prefixes contained in the *PATH* environment variable, one per line:

```
BEGIN  {
        n = split (ENVIRON["PATH"], path, ":")
        for (i = 1; i <= n; ++i)
        print path[i]
}
```

19. If there is a file named **input** containing page headers of the form:

    Page #

    and a file named **program** that contains:

```
/Page/   { $2 = n++; }
         { print }
```

then the command line:

```
awk −f program n=5 input
```

prints the file **input**, filling in page numbers starting at 5.

RATIONALE

The ISO POSIX-2 standard description is based on the new *awk*, ‘‘nawk’’, (see the referenced *The AWK Programming Language*), which introduced a number of new features to the historical *awk*:

1. New keywords: **delete**, **do**, **functin**, **return**
2. New built-in functions: **atan2**, **close**, **cos**, **gsub**, **match**, **rand**, **sin**, **srand**, **sub**, **system**
3. New predefined variables: **FNR**, **ARGC**, **ARGV**, **RSTART**, **RLENGTH**, **SUBSEP**
4. New expression operators: **?**, **:**, **,**, **^**
5. The **FS** variable and the third argument to **split**, now treated as extended regular expressions.
6. The operator precedence, changed to more closely match the C language. Two examples of code that operate differently are:

```
while ( n /= 10 > 1) ...
if (!"wk" ~ /bwk/) ...
```

Several features have been added based on newer implementations of *awk*:

- Multiple instances of −**f** *progfile* are permitted
- The new option −**v** *assignment*
- The new predefined variable **ENVIRON**
- New built-in functions **toupper**, and **tolower**
- More formatting capabilities are added to **printf** to match the ISO C standard

The overall *awk* syntax has always been based on the C language, with a few features from the shell command language and other sources. Because of this, it is not completely compatible with any other language, which has caused confusion for some users. It is not the intent of the standard developers to address such issues. IEEE Std 1003.1-200x has made a few relatively minor changes toward making the language more compatible with the C language as specified by the ISO C standard; most of these changes are based on similar changes in recent implementations, as described above. There remain several C-language conventions that are not in *awk*. One of the notable ones is the comma operator, which is commonly used to specify multiple expressions in the C language **for** statement. Also, there are various places where *awk* is more restrictive than the C language regarding the type of expression that can be used in a given context. These limitations are due to the different features that the *awk* language does provide.

Regular expressions in *awk* have been extended somewhat from historical implementations to make them a pure superset of extended regular expressions, as defined by IEEE Std 1003.1-200x (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.4, Extended Regular Expressions). The main extensions are internationalization features and interval expressions. Historical implementations of *awk* have long supported backslash escape sequences as an extension to extended regular expressions, and this extension has been retained despite inconsistency with other utilities. The number of escape sequences recognized in both extended regular expressions and strings has varied (generally increasing with time) among implementations. The set specified by IEEE Std 1003.1-200x includes most sequences known to be supported by popular implementations and by the ISO C standard. One sequence that is not supported is hexadecimal value escapes beginning with `'\x'`. This would allow values expressed in more than 9 bits to be used within *awk* as in the ISO C standard. However, because this syntax has a non-deterministic length, it does not permit the subsequent character to be a hexadecimal digit. This limitation can be dealt with in the C language by the use of lexical string concatenation. In the *awk* language, concatenation could also be a solution for strings, but not for extended regular expressions (either lexical ERE tokens or strings used dynamically as regular expressions). Because of this limitation, the feature has not been added to IEEE Std 1003.1-200x.

When a string variable is used in a context where an extended regular expression normally appears (where the lexical token ERE is used in the grammar) the string does not contain the literal slashes.

Some versions of *awk* allow the form:

```
func name(args, ... ) { statements }
```

This has been deprecated by the authors of the language, who asked that it not be included in IEEE Std 1003.1-200x.

Historical implementations of *awk* produce an error if a **next** statement is executed in a **BEGIN** action, and cause *awk* to terminate if a **next** statement is executed in an **END** action. This behavior has not been documented, and it was not believed that it was necessary to standardize it.

The specification of conversions between string and numeric values is much more detailed than in the documentation of historical implementations or in the referenced *The AWK Programming Language*. Although most of the behavior is designed to be intuitive, the details are necessary to ensure compatible behavior from different implementations. This is especially important in relational expressions since the types of the operands determine whether a string or numeric comparison is performed. From the perspective of an application writer, it is usually sufficient to expect intuitive behavior and to force conversions (by adding zero or concatenating a null string) when the type of an expression does not obviously match what is needed. The intent has been to specify historical practice in almost all cases. The one exception is that, in historical implementations, variables and constants maintain both string and numeric values after their original value is converted by any use. This means that referencing a variable or constant can have unexpected side effects. For example, with historical implementations the following program:

```
{
    a = "+2"
```

```
    b = 2
    if (NR % 2)
        c = a + b
    if (a == b)
        print "numeric comparison"
    else
        print "string comparison"
}
```

would perform a numeric comparison (and output numeric comparison) for each odd-numbered line, but perform a string comparison (and output string comparison) for each even-numbered line. IEEE Std 1003.1-200x ensures that comparisons will be numeric if necessary. With historical implementations, the following program:

```
BEGIN {
    OFMT = "%e"
    print 3.14
    OFMT = "%f"
    print 3.14
}
```

would output `"3.140000e+00"` twice, because in the second **print** statement the constant `"3.14"` would have a string value from the previous conversion. IEEE Std 1003.1-200x requires that the output of the second **print** statement be `"3.140000"`. The behavior of historical implementations was seen as too unintuitive and unpredictable.

It was pointed out that with the rules contained in early drafts, the following script would print nothing:

```
BEGIN {
    y[1.5] = 1
    OFMT = "%e"
    print y[1.5]
}
```

Therefore, a new variable, **CONVFMT**, was introduced. The **OFMT** variable is now restricted to affecting output conversions of numbers to strings and **CONVFMT** is used for internal conversions, such as comparisons or array indexing. The default value is the same as that for **OFMT**, so unless a program changes **CONVFMT** (which no historical program would do), it will receive the historical behavior associated with internal string conversions.

The POSIX *awk* lexical and syntactic conventions are specified more formally than in other sources. Again the intent has been to specify historical practice. One convention that may not be obvious from the formal grammar as in other verbal descriptions is where <newline>s are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape <newline>s between any lexical tokens. In addition, <newline>s without backslashes can follow a comma, an open brace, a logical AND operator ("&&"), a logical OR operator (`"||"`), the **do** keyword, the **else** keyword, and the closing parenthesis of an **if**, **for**, or **while** statement. For example:

```
{ print $1,
        $2 }
```

The requirement that *awk* add a trailing <newline> to the program argument text is to simplify the grammar, making it match a text file in form. There is no way for an application or test suite to determine whether a literal <newline> is added or whether *awk* simply acts as if it did.

IEEE Std 1003.1-200x requires several changes from historical implementations in order to support internationalization. Probably the most subtle of these is the use of the decimal-point character, defined by the *LC_NUMERIC* category of the locale, in representations of floating-point numbers. This locale-specific character is used in recognizing numeric input, in converting between strings and numeric values, and in formatting output. However, regardless of locale, the period character (the decimal-point character of the POSIX locale) is the decimal-point character recognized in processing *awk* programs (including assignments in command line arguments). This is essentially the same convention as the one used in the ISO C standard. The difference is that the C language includes the *setlocale*( ) function, which permits an application to modify its locale. Because of this capability, a C application begins executing with its locale set to the C locale, and only executes in the environment-specified locale after an explicit call to *setlocale*( ). However, adding such an elaborate new feature to the *awk* language was seen as inappropriate for IEEE Std 1003.1-200x. It is possible to execute an *awk* program explicitly in any desired locale by setting the environment in the shell.

The undefined behavior resulting from NULs in extended regular expressions allows future extensions for the GNU *gawk* program to process binary data.

The behavior in the case of invalid *awk* programs (including lexical, syntactic, and semantic errors) is undefined because it was considered overly limiting on implementations to specify. In most cases such errors can be expected to produce a diagnostic and a non-zero exit status. However, some implementations may choose to extend the language in ways that make use of certain invalid constructs. Other invalid constructs might be deemed worthy of a warning, but otherwise cause some reasonable behavior. Still other constructs may be very difficult to detect in some implementations. Also, different implementations might detect a given error during an initial parsing of the program (before reading any input files) while others might detect it when executing the program after reading some input. Implementors should be aware that diagnosing errors as early as possible and producing useful diagnostics can ease debugging of applications, and thus make an implementation more usable.

The unspecified behavior from using multi-character **RS** values is to allow possible future extensions based on extended regular expressions used for record separators. Historical implementations take the first character of the string and ignore the others.

Unspecified behavior when *split*(*string*,*array*,<null>) is used is to allow a proposed future extension that would split up a string into an array of individual characters.

In the context of the **getline** function, equally good arguments for different precedences of the **|** and **<** operators can be made. Historical practice has been that:

```
getline < "a" "b"
```

is parsed as:

```
( getline < "a" ) "b"
```

although many would argue that the intent was that the file **ab** should be read. However:

```
getline < "x" + 1
```

parses as:

```
getline < ( "x" + 1 )
```

Similar problems occur with the **|** version of **getline**, particularly in combination with **$**. For example:

```
$"echo hi" | getline
```

(This situation is particularly problematic when used in a **print** statement, where the **|getline** part might be a redirection of the **print**.)

Since in most cases such constructs are not (or at least should not) be used (because they have a natural ambiguity for which there is no conventional parsing), the meaning of these constructs has been made explicitly unspecified. (The effect is that a conforming application that runs into the problem must parenthesize to resolve the ambiguity.) There appeared to be few if any actual uses of such constructs.

Grammars can be written that would cause an error under these circumstances. Where backwards compatibility is not a large consideration, implementors may wish to use such grammars.

Some historical implementations have allowed some built-in functions to be called without an argument list, the result being a default argument list chosen in some ‘‘reasonable’’ way. Use of **length** as a synonym for **length($0)** is the only one of these forms that is thought to be widely known or widely used; this particular form is documented in various places (for example, most historical *awk* reference pages, although not in the referenced *The AWK Programming Language*) as legitimate practice. With this exception, default argument lists have always been undocumented and vaguely defined, and it is not at all clear how (or if) they should be generalized to user-defined functions. They add no useful functionality and preclude possible future extensions that might need to name functions without calling them. Not standardizing them seems the simplest course. The standard developers considered that **length** merited special treatment, however, since it has been documented in the past and sees possibly substantial use in historical programs. Accordingly, this usage has been made legitimate, but Issue 5 removed the obsolescent marking for XSI-conforming implementations and many otherwise conforming applications depend on this feature.

In **sub** and **gsub**, if *repl* is a string literal (the lexical token **STRING**), then two consecutive backslash characters should be used in the string to ensure a single backslash will precede the ampersand when the resultant string is passed to the function. (For example, to specify one literal ampersand in the replacement string, use **gsub**(**ERE**, `"\\&"`).)

Historically the only special character in the *repl* argument of **sub** and **gsub** string functions was the ampersand (`'&'`) character and preceding it with the backslash character was used to turn off its special meaning.

The description in the ISO POSIX-2: 1993 standard introduced behavior such that the backslash character was another special character and it was unspecified whether there were any other special characters. This description introduced several portability problems, some of which are described below, and so it has been replaced with the more historical description. Some of the problems include:

- Historically, to create the replacement string, a script could use **gsub**(**ERE**, `"\\&"`), but with the ISO POSIX-2: 1993 standard wording, it was necessary to use **gsub**(**ERE**, `"\\\\&"`). Backslash characters are doubled here because all string literals are subject to lexical analysis, which would reduce each pair of backslash characters to a single backslash before being passed to **gsub**.
- Since it was unspecified what the special characters were, for portable scripts to guarantee that characters are printed literally, each character had to be preceded with a backslash. (For example, a portable script had to use **gsub**(**ERE**, `"\\h\\i"`) to produce a replacement string of `"hi"`.)

The description for comparisons in the ISO POSIX-2: 1993 standard did not properly describe historical practice because of the way numeric strings are compared as numbers. The current rules cause the following code:

```
if (0 == "000")
    print "strange, but true"
else
    print "not true"
```

to do a numeric comparison, causing the **if** to succeed. It should be intuitively obvious that this is incorrect behavior, and indeed, no historical implementation of *awk* actually behaves this way.

To fix this problem, the definition of *numeric string* was enhanced to include only those values obtained from specific circumstances (mostly external sources) where it is not possible to determine unambiguously whether the value is intended to be a string or a numeric.

Variables that are assigned to a numeric string shall also be treated as a numeric string. (For example, the notion of a numeric string can be propagated across assignments.) In comparisons, all variables having the uninitialized value are to be treated as a numeric operand evaluating to the numeric value zero.

Uninitialized variables include all types of variables including scalars, array elements, and fields. The definition of an uninitialized value in **Variables and Special Variables** (on page 2361) is necessary to describe the value placed on uninitialized variables and on fields that are valid (for example, **< $NF**) but have no characters in them and to describe how these variables are to be used in comparisons. A valid field, such as **$1**, that has no characters in it can be obtained by from an input line of `"\t\t"` when **FS=**`'\t'`. Historically, the comparison (**$1**<10) was done numerically after evaluating **$1** to the value zero.

The phrase ‘‘... also shall have the numeric value of the numeric string’’ was removed from several sections of the ISO POSIX-2:1993 standard because is specifies an unnecessary implementation detail. It is not necessary for IEEE Std 1003.1-200x to specify that these objects be assigned two different values. It is only necessary to specify that these objects may evaluate to two different values depending on context.

The description of numeric string processing is based on the behavior of the *atof*( ) function in the ISO C standard. While it is not a requirement for an implementation to use this function, many historical implementations of *awk* do. In the ISO C standard, floating-point constants use a period as a decimal point character for the language itself, independent of the current locale, but the *atof*( ) function and the associated *strtod*( ) function use the decimal point character of the current locale when converting strings to numeric values. Similarly in *awk*, floating-point constants in an *awk* script use a period independent of the locale, but input strings use the decimal point character of the locale.

FUTURE DIRECTIONS

None.

SEE ALSO

grep, *lex*, *sed*, the System Interfaces volume of IEEE Std 1003.1-200x, *atof*( ), *setlocale*( ), *strtod*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The *awk* utility is aligned with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements. |

IEEE PASC Interpretation 1003.2 #211 is applied, adding the sentence ‘‘An occurrence of two consecutive backslashes shall be interpreted as just a single literal backslash character.’’ into the description of the **sub** string function.

NAME

basename — return non-directory portion of a pathname

SYNOPSIS

```
basename string [suffix]
```

DESCRIPTION

The *string* operand shall be treated as a pathname, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.266, Pathname. The string *string* shall be converted to the filename corresponding to the last pathname component in *string* and then the suffix string *suffix*, if present, shall be removed. This shall be done by performing actions equivalent to the following steps in order:

1. If *string* is a null string, it is unspecified whether the resulting string is `'.'` or a null string. In either case, skip steps 2 through 6.
2. If *string* is `"//"`, it is implementation-defined whether steps 3 to 6 are skipped or processed.
3. If *string* consists entirely of slash characters, *string* shall be set to a single slash character. In this case, skip steps 4 to 6.
4. If there are any trailing slash characters in *string*, they shall be removed.
5. If there are any slash characters remaining in *string*, the prefix of *string* up to and including the last slash character in *string* shall be removed.
6. If the *suffix* operand is present, is not identical to the characters remaining in *string*, and is identical to a suffix of the characters remaining in *string*, the suffix *suffix* shall be removed from *string*. Otherwise, *string* is not modified by this step. It shall not be considered an error if *suffix* is not found in *string*.

The resulting string shall be written to standard output.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

string	A string.
suffix	A string.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *basename*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *basename* utility shall write a line to the standard output in the following format:

```
"%s\n", <resulting string>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The definition of *pathname* specifies implementation-defined behavior for pathnames starting with two slash characters. Therefore, applications shall not arbitrarily add slashes to the beginning of a pathname unless they can ensure that there are more or less than two or are prepared to deal with the implementation-defined consequences.

EXAMPLES

If the string *string* is a valid pathname:

```
$(basename "string")
```

produces a filename that could be used to open the file named by *string* in the directory returned by:

```
$(dirname "string")
```

If the string *string* is not a valid pathname, the same algorithm is used, but the result need not be a valid filename. The *basename* utility is not expected to make any judgements about the validity of *string* as a pathname; it just follows the specified algorithm to produce a result string.

The following shell script compiles **/usr/src/cmd/cat.c** and moves the output to a file named **cat** in the current directory when invoked with the argument **/usr/src/cmd/cat** or with the argument **/usr/src/cmd/cat.c**:

```
c99 $(dirname "$1")/$(basename "$1" .c).c
mv a.out $(basename "$1" .c)
```

RATIONALE

The behaviors of *basename* and *dirname* have been coordinated so that when *string* is a valid pathname:

`$(basename "string")`

would be a valid filename for the file in the directory:

`$(dirname "string")`

This would not work for the early proposal versions of these utilities due to the way it specified handling of trailing slashes.

Since the definition of *pathname* specifies implementation-defined behavior for pathnames starting with two slash characters, this volume of IEEE Std 1003.1-200x specifies similar implementation-defined behavior for the *basename* and *dirname* utilities.

FUTURE DIRECTIONS

None.

SEE ALSO

dirname, Section 2.5 (on page 2235)

CHANGE HISTORY

First released in Issue 2.

Issue 6

IEEE PASC Interpretation 1003.2 #164 is applied.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

batch — schedule commands to be executed in a batch queue

SYNOPSIS

UP

```
batch
```

DESCRIPTION

The *batch* utility shall read commands from standard input and schedule them for execution in a batch queue. It shall be the equivalent of the command:

```
at −q b −m now
```

where queue *b* is a special *at* queue, specifically for batch jobs. Batch jobs shall be submitted to the batch queue with no time constraints and shall be run by the system using algorithms, based on unspecified factors, that may vary with each invocation of *batch*.

XSI Users shall be permitted to use *batch* if their name appears in the file **/usr/lib/cron/at.allow**. If that file does not exist, the file **/usr/lib/cron/at.deny** shall be checked to determine whether the user shall be denied access to *batch*. If neither file exists, only a process with the appropriate privileges shall be allowed to submit a job. If only **at.deny** exists and is empty, global usage shall be permitted. The **at.allow** and **at.deny** files shall consist of one user name per line.

OPTIONS

None.

OPERANDS

None.

STDIN

The standard input shall be a text file consisting of commands acceptable to the shell command language described in Chapter 2 (on page 2231).

INPUT FILES

XSI The text files **/usr/lib/cron/at.allow** and **/usr/lib/cron/at.deny** shall contain zero or more user names, one per line, of users who are, respectively, authorized or denied access to the *at* and *batch* utilities.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *batch*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
LC_TIME	Determine the format and contents for date and time strings written by *batch*.

XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
SHELL	Determine the name of a command interpreter to be used to invoke the at-job. If the variable is unset or null, *sh* shall be used. If it is set to a value other than a name for *sh*, the implementation shall do one of the following: use that shell; use *sh*; use the login shell from the user database; any of the preceding accompanied by a warning diagnostic about which was chosen.
TZ	Determine the timezone. The job shall be submitted for execution at the time specified by *timespec* or **−t** *time* relative to the timezone specified by the *TZ* variable. If *timespec* specifies a timezone, it overrides *TZ*. If *timespec* does not specify a timezone and *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When standard input is a terminal, prompts of unspecified format for each line of the user input described in the STDIN section may be written to standard output.

STDERR

The following shall be written to standard error when a job has been successfully submitted:

```
"job %s at %s\n", at_job_id, <date>
```

where *date* shall be equivalent in format to the output of:

```
date +"%a %b %e %T %Y"
```

The date and time written shall be adjusted so that they appear in the timezone of the user (as determined by the *TZ* variable).

Neither this, nor warning messages concerning the selection of the command interpreter, are considered a diagnostic that changes the exit status.

Diagnostic messages, if any, shall be written to standard error.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

The job shall not be scheduled.

APPLICATION USAGE

It may be useful to redirect standard output within the specified commands.

EXAMPLES

1. This sequence can be used at a terminal:

```
batch
sort < file >outfile
EOT
```

2. This sequence, which demonstrates redirecting standard error to a pipe, is useful in a command procedure (the sequence of output redirection specifications is significant):

```
batch <<! diff file1 file2 2>&1 >outfile | mailx mygroup !
```

RATIONALE

Early proposals described *batch* in a manner totally separated from *at*, even though the historical model treated it almost as a synonym for *at* −**qb**. A number of features were added to list and control batch work separately from those in *at*. Upon further reflection, it was decided that the benefit of this did not merit the change to the historical interface.

The −**m** option was included on the equivalent *at* command because it is historical practice to mail results to the submitter, even if all job-produced output is redirected. As explained in the RATIONALE for *at*, the **now** keyword submits the job for immediate execution (after scheduling delays), despite some historical systems where *at* **now** would have been considered an error.

FUTURE DIRECTIONS

None.

SEE ALSO

at

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The NAME is changed to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

bc — arbitrary-precision arithmetic language

SYNOPSIS

`bc [−l] [file ...]`

DESCRIPTION

The *bc* utility shall implement an arbitrary precision calculator. It shall take input from any files given, then read from the standard input. If the standard input and standard output to *bc* are attached to a terminal, the invocation of *bc* shall be considered to be *interactive*, causing behavioral constraints described in the following sections.

OPTIONS

The *bc* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−l	(The letter ell.) Define the math functions and initialize *scale* to 20, instead of the default zero; see the EXTENDED DESCRIPTION section.

OPERANDS

The following operand shall be supported:

file	A pathname of a text file containing *bc* program statements. After all *file*s have been read, *bc* shall read the standard input.

STDIN

See the INPUT FILES section.

INPUT FILES

Input files shall be text files containing a sequence of comments, statements, and function definitions that shall be executed as they are read.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *bc*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output of the *bc* utility shall be controlled by the program read, and consist of zero or more lines containing the value of all executed expressions without assignments. The radix and precision of the output shall be controlled by the values of the **obase** and **scale** variables; see the EXTENDED DESCRIPTION section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

Grammar

The grammar in this section and the lexical conventions in the following section shall together describe the syntax for *bc* programs. The general conventions for this style of grammar are described in Section 1.10 (on page 2220). A valid program can be represented as the non-terminal symbol **program** in the grammar. This formal syntax shall take precedence over the text syntax description.

```
%token    EOF NEWLINE STRING LETTER NUMBER

%token    MUL_OP
/*        '*', '/', '%'                           */

%token    ASSIGN_OP
/*        '=', '+=', '-=', '*=', '/=', '%=', '^=' */

%token    REL_OP
/*        '==', '<=', '>=', '!=', '<', '>'        */

%token    INCR_DECR
/*        '++', '--'                              */

%token    Define    Break    Quit    Length
/*        'define', 'break', 'quit', 'length'     */

%token    Return    For    If    While    Sqrt
/*        'return', 'for', 'if', 'while', 'sqrt'  */

%token    Scale    Ibase    Obase    Auto
/*        'scale', 'ibase', 'obase', 'auto'       */

%start    program

%%

program              : EOF
                     | input_item program
                     ;

input_item           : semicolon_list NEWLINE
                     | function
                     ;

semicolon_list       : /* empty */
                     | statement
                     | semicolon_list ';' statement
                     | semicolon_list ';'
```

```
                         ;
statement_list         : /* empty */
                       | statement
                       | statement_list NEWLINE
                       | statement_list NEWLINE statement
                       | statement_list ';'
                       | statement_list ';' statement
                       ;
statement              : expression
                       | STRING
                       | Break
                       | Quit
                       | Return
                       | Return '(' return_expression ')'
                       | For '(' expression ';'
                             relational_expression ';'
                             expression ')' statement
                       | If '(' relational_expression ')' statement
                       | While '(' relational_expression ')' statement
                       | '{' statement_list '}'
                       ;
function               : Define LETTER '(' opt_parameter_list ')'
                             '{' NEWLINE opt_auto_define_list
                             statement_list '}'
                       ;
opt_parameter_list     : /* empty */
                       | parameter_list
                       ;
parameter_list         : LETTER
                       | define_list ',' LETTER
                       ;
opt_auto_define_list : /* empty */
                       | Auto define_list NEWLINE
                       | Auto define_list ';'
                       ;
define_list            : LETTER
                       | LETTER '[' ']'
                       | define_list ',' LETTER
                       | define_list ',' LETTER '[' ']'
                       ;
opt_argument_list      : /* empty */
                       | argument_list
                       ;
argument_list          : expression
                       | LETTER '[' ']' ',' argument_list"
                       ;
```

```
relational_expression : expression
                      | expression REL_OP expression
                      ;

return_expression     : /* empty */
                      | expression
                      ;

expression            : named_expression
                      | NUMBER
                      | ’(’ expression ’)’
                      | LETTER ’(’ opt_argument_list ’)’
                      | ’−’ expression
                      | expression ’+’ expression
                      | expression ’−’ expression
                      | expression MUL_OP expression
                      | expression ’^’ expression
                      | INCR_DECR named_expression
                      | named_expression INCR_DECR
                      | named_expression ASSIGN_OP expression
                      | Length ’(’ expression ’)’
                      | Sqrt ’(’ expression ’)’
                      | Scale ’(’ expression ’)’
                      ;

named_expression      : LETTER
                      | LETTER ’[’ expression ’]’
                      | Scale
                      | Ibase
                      | Obase
                      ;
```

Lexical Conventions in bc

The lexical conventions for *bc* programs, with respect to the preceding grammar, shall be as follows:

1. Except as noted, *bc* shall recognize the longest possible token or delimiter beginning at a given point.
2. A comment shall consist of any characters beginning with the two adjacent characters `"/*"` and terminated by the next occurrence of the two adjacent characters `"*/"`. Comments shall have no effect except to delimit lexical tokens.
3. The <newline> shall be recognized as the token **NEWLINE**.
4. The token **STRING** shall represent a string constant; it shall consist of any characters beginning with the double-quote character (`’"’`) and terminated by another occurrence of the double-quote character. The value of the string is the sequence of all characters between, but not including, the two double-quote characters. All characters shall be taken literally from the input, and there is no way to specify a string containing a double-quote character. The length of the value of each string shall be limited to {BC_STRING_MAX} bytes.
5. A <blank> shall have no effect except as an ordinary character if it appears within a **STRING** token, or to delimit a lexical token other than **STRING**.

6. The combination of a backslash character immediately followed by a <newline> shall have no effect other than to delimit lexical tokens with the following exceptions:

   - It shall be interpreted as the character sequence `"\<newline>"` in **STRING** tokens.
   - It shall be ignored as part of a multi-line **NUMBER** token.

7. The token **NUMBER** shall represent a numeric constant. It shall be recognized by the following grammar:

```
NUMBER  : integer
        | '.' integer
        | integer '.'
        | integer '.' integer
        ;

integer : digit
        | integer digit
        ;

digit   : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
        | 8 | 9 | A | B | C | D | E | F
        ;
```

8. The value of a **NUMBER** token shall be interpreted as a numeral in the base specified by the value of the internal register **ibase** (described below). Each of the **digit** characters shall have the value from 0 to 15 in the order listed here, and the period character shall represent the radix point. The behavior is undefined if digits greater than or equal to the value of **ibase** appear in the token. However, note the exception for single-digit values being assigned to **ibase** and **obase** themselves, in **Operations in bc** (on page 2400).

9. The following keywords shall be recognized as tokens:

   auto **ibase** **length** **return** **while** |
   break **if** **obase** **scale** |
   define **for** **quit** **sqrt** |

10. Any of the following characters occurring anywhere except within a keyword shall be recognized as the token **LETTER**:

    `a b c d e f g h i j k l m n o p q r s t u v w x y z`

11. The following single-character and two-character sequences shall be recognized as the token **ASSIGN_OP**:

    `=    +=    −=    *=    /=    %=    ^=`

12. If an `'='` character, as the beginning of a token, is followed by a `'−'` character with no intervening delimiter, the behavior is undefined.

13. The following single-characters shall be recognized as the token **MUL_OP**:

    `*    /    %`

14. The following single-character and two-character sequences shall be recognized as the token **REL_OP**:

    `==    <=    >=    !=    <    >`

15. The following two-character sequences shall be recognized as the token **INCR_DECR**:

`++ −−`

16. The following single characters shall be recognized as tokens whose names are the character:

    `<newline> ( ) , + − ; [ ] ˆ { }`

17. The token **EOF** is returned when the end of input is reached.

Operations in bc

There are three kinds of identifiers: ordinary identifiers, array identifiers, and function identifiers. All three types consist of single lowercase letters. Array identifiers shall be followed by square brackets (`"[]"`). An array subscript is required except in an argument or auto list. Arrays are singly dimensioned and can contain up to {BC_DIM_MAX} elements. Indexing shall begin at zero so an array is indexed from 0 to {BC_DIM_MAX}−1. Subscripts shall be truncated to integers. The application shall ensure that function identifiers are followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict.

The following table summarizes the rules for precedence and associativity of all operators. Operators on the same line shall have the same precedence; rows are in order of decreasing precedence.

Table 4-3 Operators in *bc*

Operator	Associativity
`++, −−`	N/A
`unary −`	N/A
`^`	Right to left
`*, /, %`	Left to right
`+, binary −`	Left to right
`=, +=, −=, *=, /=, %=, ˆ=`	Right to left
`==, <=, >=, !=, <, >`	None

Each expression or named expression has a *scale*, which is the number of decimal digits that shall be maintained as the fractional portion of the expression.

Named expressions are places where values are stored. Named expressions shall be valid on the left side of an assignment. The value of a named expression shall be the value stored in the place named. Simple identifiers and array elements are named expressions; they have an initial value of zero and an initial scale of zero.

The internal registers **scale**, **ibase**, and **obase** are all named expressions. The scale of an expression consisting of the name of one of these registers shall be zero; values assigned to any of these registers are truncated to integers. The **scale** register shall contain a global value used in computing the scale of expressions (as described below). The value of the register **scale** is limited to 0 ≤ **scale** ≤ {BC_SCALE_MAX} and shall have a default value of zero. The **ibase** and **obase** registers are the input and output number radix, respectively. The value of **ibase** shall be limited to:

`2 ≤ ibase ≤ 16`

The value of **obase** shall be limited to:

`2 ≤ obase ≤ {BC_BASE_MAX}`

When either **ibase** or **obase** is assigned a single **digit** value from the list in **Lexical Conventions in bc** (on page 2398), the value shall be assumed in hexadecimal. (For example, **ibase**=A sets to base ten, regardless of the current **ibase** value.) Otherwise, the behavior is undefined when digits greater than or equal to the value of **ibase** appear in the input. Both **ibase** and **obase** shall have initial values of 10.

Internal computations shall be conducted as if in decimal, regardless of the input and output bases, to the specified number of decimal digits. When an exact result is not achieved, (for example, **scale**=0; 3.2 / 1) the result shall be truncated.

For all values of **obase** specified by this volume of IEEE Std 1003.1-200x, *bc* shall output numeric values by performing each of the following steps in order:

1. If the value is less than zero, a hyphen (`'−'`) character shall be output.
2. One of the following is output, depending on the numerical value:
   - If the absolute value of the numerical value is greater than or equal to one, the integer portion of the value shall be output as a series of digits appropriate to **obase** (as described below) most significant digit first. The most significant non-zero digit shall be output next, followed by each successively less significant digit.
   - If the absolute value of the numerical value is less than one but greater than zero and the scale of the numerical value is greater than zero, it is unspecified whether the character 0 is output.
   - If the numerical value is zero, the character 0 shall be output.
3. If the scale of the value is greater than zero and the numeric value is not zero, a period character shall be output, followed by a series of digits appropriate to **obase** (as described below) representing the most significant portion of the fractional part of the value. If *s* represents the scale of the value being output, the number of digits output shall be *s* if **obase** is 10, less than or equal to *s* if **obase** is greater than 10, or greater than or equal to *s* if **obase** is less than 10. For **obase** values other than 10, this should be the number of digits needed to represent a precision of 10^s.

For **obase** values from 2 to 16, valid digits are the first **obase** of the single characters:

```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

which represent the values zero to 15, inclusive, respectively.

For bases greater than 16, each digit shall be written as a separate multi-digit decimal number. Each digit except the most significant fractional digit shall be preceded by a single <space>. For bases from 17 to 100, *bc* shall write two-digit decimal numbers; for bases from 101 to 1 000, three-digit decimal strings, and so on. For example, the decimal number 1 024 in base 25 would be written as:

```
Δ01Δ15Δ24
```

in base 125, as:

```
Δ008Δ024
```

Very large numbers shall be split across lines with 70 characters per line in the POSIX locale; other locales may split at different character boundaries. Lines that are continued shall end with a backslash (`'\'`).

A function call shall consist of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument shall be specified by the array name followed by empty square brackets. All function arguments shall be passed by value. As a result, changes made to the formal parameters shall have no effect on the actual arguments. If the function terminates by executing a **return** statement, the value of the function shall be the value of the expression in the parentheses of the **return** statement or shall be zero if no expression is provided or if there is no **return** statement.

The result of **sqrt**(*expression*) shall be the square root of the expression. The result shall be truncated in the least significant decimal place. The scale of the result shall be the scale of the expression or the value of **scale**, whichever is larger.

The result of **length**(*expression*) shall be the total number of significant decimal digits in the expression. The scale of the result shall be zero.

The result of **scale**(*expression*) shall be the scale of the expression. The scale of the result shall be zero.

A numeric constant shall be an expression. The scale shall be the number of digits that follow the radix point in the input representing the constant, or zero if no radix point appears.

The sequence (*expression*) shall be an expression with the same value and scale as *expression*. The parentheses can be used to alter the normal precedence.

The semantics of the unary and binary operators are as follows:

–expression
The result shall be the negative of the *expression*. The scale of the result shall be the scale of *expression*.

The unary increment and decrement operators shall not modify the scale of the named expression upon which they operate. The scale of the result shall be the scale of that named expression.

++named-expression
The named expression shall be incremented by one. The result shall be the value of the named expression after incrementing.

– –named-expression
The named expression shall be decremented by one. The result shall be the value of the named expression after decrementing.

named-expression++
The named expression shall be incremented by one. The result shall be the value of the named expression before incrementing.

named-expression– –
The named expression shall be decremented by one. The result shall be the value of the named expression before decrementing.

The exponentiation operator, circumflex (`'^'`), shall bind right to left.

expression^expression
The result shall be the first *expression* raised to the power of the second *expression*. If the second expression is not an integer, the behavior is undefined. If *a* is the scale of the left expression and *b* is the absolute value of the right expression, the scale of the result shall be:

```
if b >= 0 min(a * b, max(scale, a)) if b < 0 scale
```

The multiplicative operators (`'*'`, `'/'`, `'%'`) shall bind left to right.

*expression*expression*
The result shall be the product of the two expressions. If *a* and *b* are the scales of the two expressions, then the scale of the result shall be:

```
min(a+b,max(scale,a,b))
```

expression / *expression*
The result shall be the quotient of the two expressions. The scale of the result shall be the value of **scale**.

expression%*expression*
For expressions *a* and *b*, *a*%*b* shall be evaluated equivalent to the steps:

1. Compute *a* / *b* to current scale.
2. Use the result to compute:

```
a - (a / b) * b
```

to scale:

```
max(scale + scale(b), scale(a))
```

The scale of the result shall be:

```
max(scale + scale(b), scale(a))
```

When **scale** is zero, the `'%'` operator is the mathematical remainder operator.

The additive operators (`'+'`, `'-'`) shall bind left to right.

expression+*expression*
The result shall be the sum of the two expressions. The scale of the result shall be the maximum of the scales of the expressions.

expression−*expression*
The result shall be the difference of the two expressions. The scale of the result shall be the maximum of the scales of the expressions.

The assignment operators (`'='`, `"+="`, `"-="`, `"*="`, `"/="`, `"%="`, `"^="`) shall bind right to left.

named-expression=*expression*
This expression shall result in assigning the value of the expression on the right to the named expression on the left. The scale of both the named expression and the result shall be the scale of *expression*.

The compound assignment forms:

```
named-expression <operator>= expression
```

shall be equivalent to:

```
named-expression=named-expression <operator> expression
```

except that the *named-expression* shall be evaluated only once.

Unlike all other operators, the relational operators (`'<'`, `'>'`, `"<="`, `">="`, `"=="`, `"!="`) shall be only valid as the object of an **if**, **while**, or inside a **for** statement.

expression1<*expression2*
The relation shall be true if the value of *expression1* is strictly less than the value of *expression2*.

expression1>*expression2*
The relation shall be true if the value of *expression1* is strictly greater than the value of *expression2*.

expression1<=*expression2*
The relation shall be true if the value of *expression1* is less than or equal to the value of *expression2*.

expression1>=*expression2*
The relation shall be true if the value of *expression1* is greater than or equal to the value of *expression2*.

expression1= =*expression2*
The relation shall be true if the values of *expression1* and *expression2* are equal.

expression1!=*expression2*
The relation shall be true if the values of *expression1* and *expression2* are unequal.

There are only two storage classes in *bc*, global and automatic (local). Only identifiers that are local to a function need be declared with the **auto** command. The arguments to a function shall be local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as auto shall be allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. Auto arrays shall be specified by the array name followed by empty square brackets. On entry to a function, the old values of the names that appear as parameters and as automatic variables shall be pushed onto a stack. Until the function returns, reference to these names shall refer only to the new values.

References to any of these names from other functions that are called from this function also refer to the new value until one of those functions uses the same name for a local variable.

When a statement is an expression, unless the main operator is an assignment, execution of the statement shall write the value of the expression followed by a <newline>.

When a statement is a string, execution of the statement shall write the value of the string.

Statements separated by semicolons or <newline>s shall be executed sequentially. In an interactive invocation of *bc*, each time a <newline> is read that satisfies the grammatical production:

```
input_item : semicolon_list NEWLINE
```

the sequential list of statements making up the **semicolon_list** shall be executed immediately and any output produced by that execution shall be written without any delay due to buffering.

In an **if** statement (**if**(*relation*) *statement*), the *statement* shall be executed if the relation is true.

The **while** statement (**while**(*relation*) *statement*) implements a loop in which the *relation* is tested; each time the *relation* is true, the *statement* shall be executed and the *relation* retested. When the *relation* is false, execution shall resume after *statement*.

A **for** statement(**for**(*expression*; *relation*; *expression*) *statement*) shall be the same as:

```
first-expression
while (relation) {
    statement
    last-expression
}
```

The application shall ensure that all three expressions are present.

The **break** statement shall cause termination of a **for** or **while** statement.

The **auto** statement (**auto** *identifier* **[**,*identifier***]** ...) shall cause the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers shall be specified by following the array name by empty square brackets. The application shall ensure that the **auto** statement is the first statement in a function definition.

A **define** statement:

```
define LETTER ( opt_parameter_list ) {
    opt_auto_define_list
    statement_list
}
```

defines a function named **LETTER**. If a function named **LETTER** was previously defined, the **define** statement shall replace the previous definition. The expression:

```
LETTER ( opt_argument_list )
```

shall invoke the function named **LETTER**. The behavior is undefined if the number of arguments in the invocation does not match the number of parameters in the definition. Functions shall be defined before they are invoked. A function shall be considered to be defined within its own body, so recursive calls are valid. The values of numeric constants within a function shall be interpreted in the base specified by the value of the **ibase** register when the function is invoked.

The **return** statements (**return** and **return**(*expression*)) shall cause termination of a function, popping of its auto variables, and specification of the result of the function. The first form shall be equivalent to **return**(0). The value and scale of the result returned by the function shall be the value and scale of the expression returned.

The **quit** statement (**quit**) shall stop execution of a *bc* program at the point where the statement occurs in the input, even if it occurs in a function definition, or in an **if**, **for**, or **while** statement.

The following functions shall be defined when the **–l** option is specified:

s(*expression*)
Sine of argument in radians.

c(*expression*)
Cosine of argument in radians.

a(*expression*)
Arctangent of argument.

l(*expression*)
Natural logarithm of argument.

e(*expression*)
Exponential function of argument.

j(*expression*, *expression*)
Bessel function of integer order.

The scale of the result returned by these functions shall be the value of the **scale** register at the time the function is invoked. The value of the **scale** register after these functions have completed their execution shall be the same value it had upon invocation. The behavior is undefined if any of these functions is invoked with an argument outside the domain of the mathematical function.

EXIT STATUS

The following exit values shall be returned:

0	All input files were processed successfully.

unspecified An error occurred.

CONSEQUENCES OF ERRORS

If any *file* operand is specified and the named file cannot be accessed, *bc* shall write a diagnostic message to standard error and terminate without any further action.

In an interactive invocation of *bc*, the utility should print an error message and recover following any error in the input. In a non-interactive invocation of *bc*, invalid input causes undefined behavior.

APPLICATION USAGE

Automatic variables in *bc* do not work in exactly the same way as in either C or PL/1.

For historical reasons, the exit status from *bc* cannot be relied upon to indicate that an error has occurred. Returning zero after an error is possible. Therefore, *bc* should be used primarily by interactive users (who can react to error messages) or by application programs that can somehow validate the answers returned as not including error messages.

The *bc* utility always uses the period (`'.'`) character to represent a radix point, regardless of any decimal-point character specified as part of the current locale. In languages like C or *awk*, the period character is used in program source, so it can be portable and unambiguous, while the locale-specific character is used in input and output. Because there is no distinction between source and input in *bc*, this arrangement would not be possible. Using the locale-specific character in *bc*'s input would introduce ambiguities into the language; consider the following example in a locale with a comma as the decimal-point character:

```
define f(a,b) {
    ...
}
...

f(1,2,3)
```

Because of such ambiguities, the period character is used in input. Having input follow different conventions from output would be confusing in either pipeline usage or interactive usage, so the period is also used in output.

EXAMPLES

In the shell, the following assigns an approximation of the first ten digits of 'π' to the variable *x*:

```
x=$(printf "%s\n" 'scale = 10; 104348/33215' | bc)
```

The following *bc* program prints the same approximation of 'π', with a label, to standard output:

```
scale = 10
"pi equals "
104348 / 33215
```

The following defines a function to compute an approximate value of the exponential function (note that such a function is predefined if the **–l** option is specified):

```
scale = 20
define e(x){
    auto a, b, c, i, s
    a = 1
    b = 1
    s = 1
```

```
    for (i = 1; 1 == 1; i++){
        a = a*x
        b = b*i
        c = a/b
        if (c == 0) {
             return(s)
        }
        s = s+c
    }
}
```

The following prints approximate values of the exponential function of the first ten integers:

```
for (i = 1; i <= 10; ++i) {
    e(i)
}
```

RATIONALE

The *bc* utility is implemented historically as a front-end processor for *dc*; *dc* was not selected to be part of this volume of IEEE Std 1003.1-200x because *bc* was thought to have a more intuitive programmatic interface. Current implementations that implement *bc* using *dc* are expected to be compliant.

The exit status for error conditions has been left unspecified for several reasons:

- The *bc* utility is used in both interactive and non-interactive situations. Different exit codes may be appropriate for the two uses.
- It is unclear when a non-zero exit should be given; divide-by-zero, undefined functions, and syntax errors are all possibilities.
- It is not clear what utility the exit status has.
- In the 4.3 BSD, System V, and Ninth Edition implementations, *bc* works in conjunction with *dc*. The *dc* utility is the parent, *bc* is the child. This was done to cleanly terminate *bc* if *dc* aborted.

The decision to have *bc* exit upon encountering an inaccessible input file is based on the belief that *bc file1 file2* is used most often when at least *file1* contains data/function declarations/initializations. Having *bc* continue with prerequisite files missing is probably not useful. There is no implication in the CONSEQUENCES OF ERRORS section that *bc* must check all its files for accessibility before opening any of them.

There was considerable debate on the appropriateness of the language accepted by *bc*. Several reviewers preferred to see either a pure subset of the C language or some changes to make the language more compatible with C. While the *bc* language has some obvious similarities to C, it has never claimed to be compatible with any version of C. An interpreter for a subset of C might be a very worthwhile utility, and it could potentially make *bc* obsolete. However, no such utility is known in historical practice, and it was not within the scope of this volume of IEEE Std 1003.1-200x to define such a language and utility. If and when they are defined, it may be appropriate to include them in a future version of this volume of IEEE Std 1003.1-200x. This left the following alternatives:

1. Exclude any calculator language from this volume of IEEE Std 1003.1-200x.

   The consensus of the standard developers was that a simple programmatic calculator language is very useful for both applications and interactive users. The only arguments for excluding any calculator were that it would become obsolete if and when a C-compatible one emerged, or that the absence would encourage the development of such a C-compatible one. These arguments did not sufficiently address the needs of current application writers.

2. Standardize the historical *dc*, possibly with minor modifications.

   The consensus of the standard developers was that *dc* is a fundamentally less usable language and that that would be far too severe a penalty for avoiding the issue of being similar to but incompatible with C.

3. Standardize the historical *bc*, possibly with minor modifications.

   This was the approach taken. Most of the proponents of changing the language would not have been satisfied until most or all of the incompatibilities with C were resolved. Since most of the changes considered most desirable would break historical applications and require significant modification to historical implementations, almost no modifications were made. The one significant modification that was made was the replacement of the historical *bc* assignment operators `"=+"`, and so on, with the more modern `"+="`, and so on. The older versions are considered to be fundamentally flawed because of the lexical ambiguity in uses like a=−1.

   In order to permit implementations to deal with backwards compatibility as they see fit, the behavior of this one ambiguous construct was made undefined. (At least three implementations have been known to support this change already, so the degree of change involved should not be great.)

The `'%'` operator is the mathematical remainder operator when **scale** is zero. The behavior of this operator for other values of **scale** is from historical implementations of *bc*, and has been maintained for the sake of historical applications despite its non-intuitive nature.

Historical implementations permit setting **ibase** and **obase** to a broader range of values. This includes values less than 2, which were not seen as sufficiently useful to standardize. These implementations do not interpret input properly for values of **ibase** that are greater than 16. This is because numeric constants are recognized syntactically, rather than lexically, as described in this volume of IEEE Std 1003.1-200x. They are built from lexical tokens of single hexadecimal digits and periods. Since <blank>s between tokens are not visible at the syntactic level, it is not possible to recognize the multi-digit ‘‘digits’’ used in the higher bases properly. The ability to recognize input in these bases was not considered useful enough to require modifying these implementations. Note that the recognition of numeric constants at the syntactic level is not a problem with conformance to this volume of IEEE Std 1003.1-200x, as it does not impact the behavior of conforming applications (and correct *bc* programs). Historical implementations also accept input with all of the digits `'0'`−`'9'` and `'A'`−`'F'` regardless of the value of **ibase**; since digits with value greater than or equal to **ibase** are not really appropriate, the behavior when they appear is undefined, except for the common case of:

```
ibase=8;
    /* Process in octal base. */
...
ibase=A
    /* Restore decimal base. */
```

In some historical implementations, if the expression to be written is an uninitialized array element, a leading <space> and/or up to four leading 0 characters may be output before the character zero. This behavior is considered a bug; it is unlikely that any currently conforming application relies on:

```
echo 'b[3]' | bc
```

returning 00000 rather than 0.

Exact calculation of the number of fractional digits to output for a given value in a base other than 10 can be computationally expensive. Historical implementations use a faster approximation, and this is permitted. Note that the requirements apply only to values of **obase** that this volume of IEEE Std 1003.1-200x requires implementations to support (in particular, not to 1, 0, or negative bases, if an implementation supports them as an extension).

Historical implementations of *bc* did not allow array parameters to be passed as the last parameter to a function. New implementations are encouraged to remove this restriction even though it is not required by the grammar.

FUTURE DIRECTIONS

None.

SEE ALSO

awk

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

Updated to align with the IEEE P1003.2b draft standard, which included resolution of several interpretations of the ISO POSIX-2:1993 standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

bg — run jobs in the background

SYNOPSIS

UP

```
bg [job_id ...]
```

DESCRIPTION

If job control is enabled (see the description of *set* **–m**), the *bg* utility shall resume suspended jobs from the current environment (see Section 2.12 (on page 2263)) by running them as background jobs. If the job specified by *job_id* is already a running background job, the *bg* utility shall have no effect and shall exit successfully.

Using *bg* to place a job into the background shall cause its process ID to become ‘‘known in the current shell execution environment’’, as if it had been started as an asynchronous list; see Section 2.9.3.1 (on page 2252).

OPTIONS

None.

OPERANDS

The following operand shall be supported:

job_id	Specify the job to be resumed as a background job. If no *job_id* operand is given, the most recently suspended job shall be used. The format of *job_id* is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.203, Job Control Job ID.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *bg*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output of *bg* shall consist of a line in the format:

```
"[%d] %s\n", <job-number>, <command>
```

where the fields are as follows:

<*job-number*>	A number that can be used to identify the job to the *wait*, *fg*, and *kill* utilities. Using these utilities, the job can be identified by prefixing the job number with `'%'`.
<*command*>	The associated command that was given to the shell.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If job control is disabled, the *bg* utility shall exit with an error and no job shall be placed in the background.

APPLICATION USAGE

A job is generally suspended by typing the SUSP character (<control>-Z on most systems); see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface. At that point, *bg* can put the job into the background. This is most effective when the job is expecting no terminal input and its output has been redirected to non-terminal files. A background job can be forced to stop when it has terminal output by issuing the command:

```
stty tostop
```

A background job can be stopped with the command:

```
kill −s stop job ID
```

The *bg* utility does not work as expected when it is operating in its own utility execution environment because that environment has no suspended jobs. In the following examples:

```
... | xargs bg
(bg)
```

each *bg* operates in a different environment and does not share its parent shell's understanding of jobs. For this reason, *bg* is generally implemented as a shell regular built-in.

EXAMPLES

None.

RATIONALE

The extensions to the shell specified in this volume of IEEE Std 1003.1-200x have mostly been based on features provided by the KornShell. The job control features provided by *bg*, *fg*, and *jobs* are also based on the KornShell. The standard developers examined the characteristics of the C shell versions of these utilities and found that differences exist. Despite widespread use of the C shell, the KornShell versions were selected for this volume of IEEE Std 1003.1-200x to maintain a degree of uniformity with the rest of the KornShell features selected (such as the very popular command line editing features).

The *bg* utility is expected to wrap its output if the output exceeds the number of display columns.

FUTURE DIRECTIONS

None.

SEE ALSO

fg, *kill*, *jobs*, *wait*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The JC margin marker on the SYNOPSIS is removed since support for Job Control is mandatory in this issue. This is a FIPS requirement.

NAME

c99 — compile standard C programs

SYNOPSIS

CD

```
c99 [−c][−D name[=value]]...[−E][−g][−I directory] ... [−L directory]
    ... [−o outfile][−Ooptlevel][−s][−U name]... operand ...
```

DESCRIPTION

The *c99* utility is an interface to the standard C compilation system; it shall accept source code conforming to the ISO C standard. The system conceptually consists of a compiler and link editor. The files referenced by *operand*s shall be compiled and linked to produce an executable file. (It is unspecified whether the linking occurs entirely within the operation of *c99*; some implementations may produce objects that are not fully resolved until the file is executed.)

If the **−c** option is specified, for all pathname operands of the form *file***.c**, the files:

```
$(basename pathname .c).o
```

shall be created as the result of successful compilation. If the **−c** option is not specified, it is unspecified whether such **.o** files are created or deleted for the *file***.c** operands.

If there are no options that prevent link editing (such as **−c** or **−E**), and all operands compile and link without error, the resulting executable file shall be written according to the **−o** *outfile* option (if present) or to the file **a.out**.

The executable file shall be created as specified in Section 1.7.1.4 (on page 2204), except that the file permission bits shall be set to:

S_IRWXO | S_IRWXG | S_IRWXU

and the bits specified by the *umask* of the process shall be cleared.

OPTIONS

The *c99* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that:

- The **−l** *library* operands have the format of options, but their position within a list of operands affects the order in which libraries are searched.
- The order of specifying the **−I** and **−L** options is significant.
- Conforming applications shall specify each option separately; that is, grouping option letters (for example, **−cO**) need not be recognized by all implementations.

The following options shall be supported:

−c Suppress the link-edit phase of the compilation, and do not remove any object files that are produced.

−g Produce symbolic information in the object or executable files; the nature of this information is unspecified, and may be modified by implementation-defined interactions with other options.

−s Produce object or executable files, or both, from which symbolic and other information not required for proper execution using the *exec* family defined in the System Interfaces volume of IEEE Std 1003.1-200x, has been removed (stripped). If both **−g** and **−s** options are present, the action taken is unspecified.

−o *outfile* Use the pathname *outfile*, instead of the default **a.out**, for the executable file produced. If the **−o** option is present with **−c** or **−E**, the result is unspecified.

–D *name*[=*value*]
Define *name* as if by a C-language **#define** directive. If no =*value* is given, a value of 1 shall be used. The **–D** option has lower precedence than the **–U** option. That is, if *name* is used in both a **–U** and a **–D** option, *name* shall be undefined regardless of the order of the options. Additional implementation-defined *name*s may be provided by the compiler. Implementations shall support at least 2 048 bytes of **–D** definitions and 256 *names*.

–E Copy C-language source files to standard output, expanding all preprocessor directives; no compilation shall be performed. If any operand is not a text file, the effects are unspecified.

–I *directory* Change the algorithm for searching for headers whose names are not absolute pathnames to look in the directory named by the *directory* pathname before looking in the usual places. Thus, headers whose names are enclosed in double-quotes (`" "`) shall be searched for first in the directory of the file with the **#include** line, then in directories named in **–I** options, and last in the usual places. For headers whose names are enclosed in angle brackets (`"< >"`), the header shall be searched for only in directories named in **–I** options and then in the usual places. Directories named in **–I** options shall be searched in the order specified. Implementations shall support at least ten instances of this option in a single *c99* command invocation.

–L *directory* Change the algorithm of searching for the libraries named in the **–l** objects to look in the directory named by the *directory* pathname before looking in the usual places. Directories named in **–L** options shall be searched in the order specified. Implementations shall support at least ten instances of this option in a single *c99* command invocation. If a directory specified by a **–L** option contains files named **libc.a**, **libm.a**, **libl.a**, or **liby.a**, the results are unspecified.

–O *optlevel* Specify the level of code optimization. If the *optlevel* option-argument is the digit `'0'`, all special code optimizations shall be disabled. If it is the digit `'1'`, the nature of the optimization is unspecified. If the **–O** option is omitted, the nature of the system's default optimization is unspecified. It is unspecified whether code generated in the presence of the **–O** 0 option is the same as that generated when **–O** is omitted. Other *optlevel* values may be supported.

–U *name* Remove any initial definition of *name*.

Multiple instances of the **–D**, **–I**, **–U**, and **–L** options can be specified.

OPERANDS

An *operand* is either in the form of a pathname or the form **–l** *library*. The application shall ensure that at least one operand of the pathname form is specified. The following operands shall be supported:

file.**c** A C-language source file to be compiled and optionally linked. The application shall ensure that the operand is of this form if the **–c** option is used.

file.**a** A library of object files typically produced by the *ar* utility, and passed directly to the link editor. Implementations may recognize implementation-defined suffixes other than **.a** as denoting object file libraries.

file.**o** An object file produced by *c99* **–c** and passed directly to the link editor. Implementations may recognize implementation-defined suffixes other than **.o** as denoting object files.

The processing of other files is implementation-defined.

–l library (The letter ell.) Search the library named:

`lib`*`library`*`.a`

A library shall be searched when its name is encountered, so the placement of a **–l** operand is significant. Several standard libraries can be specified in this manner, as described in the EXTENDED DESCRIPTION section. Implementations may recognize implementation-defined suffixes other than **.a** as denoting libraries.

STDIN

Not used.

INPUT FILES

The input file shall be one of the following: a text file containing a C-language source program, an object file in the format produced by *c99* **–c**, or a library of object files, in the format produced by archiving zero or more object files, using *ar*. Implementations may supply additional utilities that produce files in these formats. Additional input file formats are implementation-defined.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *c99*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

TMPDIR Provide a pathname that should override the default directory for temporary files, if any. XSI On XSI-conforming systems, provide a pathname that shall override the default directory for temporary files, if any.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If more than one *file* operand ending in **.c** (or possibly other unspecified suffixes) is given, for each such file:

`"%s:\n", <`*`file`*`>`

may be written. These messages, if written, shall precede the processing of each input file; they shall not be written to the standard output if they are written to the standard error, as described in the STDERR section.

If the **–E** option is specified, the standard output shall be a text file that represents the results of the preprocessing stage of the language; it may contain extra information appropriate for subsequent compilation passes.

STDERR

The standard error shall be used only for diagnostic messages. If more than one *file* operand ending in **.c** (or possibly other unspecified suffixes) is given, for each such file:

```
"%s:\n", <file>
```

may be written to allow identification of the diagnostic and warning messages with the appropriate input file. These messages, if written, shall precede the processing of each input file; they shall not be written to the standard error if they are written to the standard output, as described in the STDOUT section.

This utility may produce warning messages about certain conditions that do not warrant returning an error (non-zero) exit value.

OUTPUT FILES

Object files or executable files or both are produced in unspecified formats.

EXTENDED DESCRIPTION

Standard Libraries

The *c99* utility shall recognize the following **–l** operands for standard libraries:

–l c	This operand shall make visible all functions referenced in the System Interfaces volume of IEEE Std 1003.1-200x, with the possible exception of those functions listed as residing in **<aio.h>**, **<arpa/inet.h>**, **<math.h>**, **<mqueue.h>**, **<netdb.h>**, **<netinet/in.h>**, **<pthread.h>**, **<sched.h>**, **<semaphore.h>**, **<spawn.h>**, **<sys/socket.h>**, *pthread_kill*( ), and *pthread_sigmask*( ) in **<signal.h>**, functions marked as extensions other than as part of the MF or MPR extensions in **<sys/mman.h>**, functions marked as ADV in **<fcntl.h>**, and functions marked as CS, CPT, and TMR in **<time.h>**. This operand shall not be required to be present to cause a search of this library.
–l l	This operand shall make visible all functions required by the C-language output of *lex* that are not made available through the **–l c** operand.
–l pthread	This operand shall make visible all functions referenced in **<pthread.h>** and *pthread_kill*( ) and *pthread_sigmask*( ) referenced in **<signal.h>**. An implementation may search this library in the absence of this operand.
–l m	This operand shall make visible all functions referenced in **<math.h>**. An implementation may search this library in the absence of this operand.
–l rt	This operand shall make visible all functions referenced in **<aio.h>**, **<mqueue.h>**, **<sched.h>**, **<semaphore.h>**, and **<spawn.h>**, functions marked as extensions other than as part of the MF or MPR extensions in **<sys/mman.h>**, functions marked as ADV in **<fcntl.h>**, and functions marked as CS, CPT, and TMR in **<time.h>**. An implementation may search this library in the absence of this operand.
–l trace	This operand shall make visible all functions referenced in **<trace.h>**. An implementation may search this library in the absence of this operand.
–l xnet	This operand makes visible all functions referenced in **<arpa/inet.h>**, **<netdb.h>**, **<netinet/in.h>**, and **<sys/socket.h>**. An implementation may search this library in the absence of this operand.
–l y	This operand shall make visible all functions required by the C-language output of *yacc* that are not made available through the **–l c** operand.

In the absence of options that inhibit invocation of the link editor, such as **−c** or **−E**, the *c99* utility shall cause the equivalent of a **−l c** operand to be passed to the link editor as the last **−l** operand, causing it to be searched after all other object files and libraries are loaded.

It is unspecified whether the libraries **libc.a**, **libm.a**, **librt.a**, **libpthread.a**, **libl.a**, **liby.a**, or **libxnet** exist as regular files. The implementation may accept as **−l** operands names of objects that do not exist as regular files.

External Symbols

The C compiler and link editor shall support the significance of external symbols up to a length of at least 31 bytes; the action taken upon encountering symbols exceeding the implementation-defined maximum symbol length is unspecified.

The compiler and link editor shall support a minimum of 511 external symbols per source or object file, and a minimum of 4 095 external symbols in total. A diagnostic message shall be written to the standard output if the implementation-defined limit is exceeded; other actions are unspecified.

Programming Environments

All implementations shall support one of the following programming environments as a default. Implementations may support more than one of the following programming environments. Applications can use *sysconf*( ) or *getconf* to determine which programming environments are supported.

Table 4-4 Programming Environments: Type Sizes

Programming Environment *getconf* **Name**	**Bits in int**	**Bits in long**	**Bits in pointer**	**Bits in off_t**
_POSIX_V6_ILP32_OFF32	32	32	32	32
_POSIX_V6_ILP32_OFFBIG	32	32	32	≥64
_POSIX_V6_LP64_OFF64	32	64	64	64
_POSIX_V6_LPBIG_OFFBIG	≥32	≥64	≥64	≥64

All implementations shall support one or more environments where the widths of the following types are no greater than the width of type **long**:

blksize_t, **cc_t**, **mode_t**, **nfds_t**, **pid_t**, **ptrdiff_t**, **size_t**, **speed_t**, **ssize_t**, **suseconds_t**, **tcflag_t**, **useconds_t**, **wchar_t**, **wint_t**

The executable files created when these environments are selected shall be in a proper format for execution by the *exec* family of functions. Each environment may be one of the ones in Table 4-4, or it may be another environment. The names for the environments that meet this requirement shall be output by a *getconf* command using the _POSIX_V6_WIDTH_RESTRICTED_ENVS argument. If more than one environment meets the requirement, the names of all such environments shall be output on separate lines. Any of these names can then be used in a subsequent *getconf* command to obtain the flags specific to that environment with the following suffixes added as appropriate:

_CFLAGS	To get the C compiler flags.
_LDFLAGS	To get the linker/loader flags.
_LIBS	To get the libraries.

This requirement may be removed in a future version of IEEE Std 1003.1.

When this utility processes a file containing a function called *main*( ), it shall be defined with a return type equivalent to **int**. Using return from *main*( ) shall be equivalent (other than with respect to language scope issues) to calling *exit*( ) with the returned value. Reaching the end of *main*( ) shall be equivalent to calling *exit*(0). The implementation shall not declare a prototype for this function.

Implementations provide configuration strings for C compiler flags, linker/loader flags, and libraries for each supported environment. When an application needs to use a specific programming environment rather than the implementation default programming environment while compiling, the application shall first verify that the implementation supports the desired environment. If the desired programming environment is supported, the application shall then invoke *c99* with the appropriate C compiler flags as the first options for the compile, the appropriate linker/loader flags after any other options but before any operands, and the appropriate libraries at the end of the operands.

Conforming applications shall not attempt to link together object files compiled for different programming models. Applications shall also be aware that binary data placed in shared memory or in files might not be recognized by applications built for other programming models.

Table 4-5 Programming Environments: c99 and cc Arguments

Programming Environment *getconf* **Name**	**Use**	*c99* **and** *cc* **Arguments** *getconf* **Name**
_POSIX_V6_ILP32_OFF32	C Compiler Flags	POSIX_V6_ILP32_OFF32_CFLAGS
	Linker/Loader Flags	POSIX_V6_ILP32_OFF32_LDFLAGS
	Libraries	POSIX_V6_ILP32_OFF32_LIBS
_POSIX_V6_ILP32_OFFBIG	C Compiler Flags	POSIX_V6_ILP32_OFFBIG_CFLAGS
	Linker/Loader Flags	POSIX_V6_ILP32_OFFBIG_LDFLAGS
	Libraries	POSIX_V6_ILP32_OFFBIG_LIBS
_POSIX_V6_LP64_OFF64	C Compiler Flags	POSIX_V6_LP64_OFF64_CFLAGS
	Linker/Loader Flags	POSIX_V6_LP64_OFF64_LDFLAGS
	Libraries	POSIX_V6_LP64_OFF64_LIBS
_POSIX_V6_LPBIG_OFFBIG	C Compiler Flags	POSIX_V6_LPBIG_OFFBIG_CFLAGS
	Linker/Loader Flags	POSIX_V6_LPBIG_OFFBIG_LDFLAGS
	Libraries	POSIX_V6_LPBIG_OFFBIG_LIBS

EXIT STATUS

The following exit values shall be returned:

0	Successful compilation or link edit.
>0	An error occurred.

CONSEQUENCES OF ERRORS

When *c99* encounters a compilation error that causes an object file not to be created, it shall write a diagnostic to standard error and continue to compile other source code operands, but it shall not perform the link phase and return a non-zero exit status. If the link edit is unsuccessful, a diagnostic message shall be written to standard error and *c99* exits with a non-zero status. A conforming application shall rely on the exit status of *c99*, rather than on the existence or mode of the executable file.

APPLICATION USAGE

Since the *c99* utility usually creates files in the current directory during the compilation process, it is typically necessary to run the *c99* utility in a directory in which a file can be created.

On systems providing POSIX Conformance (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance), *c99* is required only with the C-Language Development option; XSI-conformant systems always provide *c99*.

Some historical implementations have created **.o** files when **–c** is not specified and more than one source file is given. Since this area is left unspecified, the application cannot rely on **.o** files being created, but it also must be prepared for any related **.o** files that already exist being deleted at the completion of the link edit.

Some historical implementations have permitted **–L** options to be interspersed with **–l** operands on the command line. For an application to compile consistently on systems that do not behave like this, it is necessary for a conforming application to supply all **–L** options before any of the **–l** options.

There is the possible implication that if a user supplies versions of the standard functions (before they would be encountered by an implicit **–l c** or explicit **–l m**), that those versions would be used in place of the standard versions. There are various reasons this might not be true (functions defined as macros, manipulations for clean name space, and so on), so the existence of files named in the same manner as the standard libraries within the **–L** directories is explicitly stated to produce unspecified behavior.

All of the functions specified in the System Interfaces volume of IEEE Std 1003.1-200x may be made visible by implementations when the Standard C Library is searched. Conforming applications must explicitly request searching the other standard libraries when functions made visible by those libraries are used.

EXAMPLES

1. The following usage example compiles **foo.c** and creates the executable file **foo**:

   ```
   c99 –o foo foo.c
   ```

   The following usage example compiles **foo.c** and creates the object file **foo.o**:

   ```
   c99 –c foo.c
   ```

   The following usage example compiles **foo.c** and creates the executable file **a.out**:

   ```
   c99 foo.c
   ```

   The following usage example compiles **foo.c**, links it with **bar.o**, and creates the executable file **a.out**. It also creates and leaves **foo.o**:

   ```
   c99 foo.c bar.o
   ```

2. The following example shows how an application using threads interfaces can test for support of and use a programming environment supporting 32-bit **int**, **long**, and **pointer** types and an **off_t** type using at least 64 bits:

   ```
   if [ $(getconf _POSIX_V6_ILP32_OFFBIG) != "–1" ]
   then
       c99 $(getconf POSIX_V6_ILP32_OFFBIG_CFLAGS) –D_XOPEN_SOURCE=600 \
           $(getconf POSIX_V6_ILP32_OFFBIG_LDFLAGS) foo.c –o foo \
           $(getconf POSIX_V6_ILP32_OFFBIG_LIBS) –l pthread
   else
       echo ILP32_OFFBIG programming environment not supported
   ```

```
    exit 1
fi
```

3. The following examples clarify the use and interactions of –**L** options and –**l** operands.

   Consider the case in which module **a.c** calls function *f*( ) in library **libQ.a**, and module **b.c** calls function *g*( ) in library **libp.a**. Assume that both libraries reside in **/a/b/c**. The command line to compile and link in the desired way is:

   ```
   c99 –L /a/b/c main.o a.c –l Q b.c –l p
   ```

   In this case the –**l Q** operand need only precede the first –**l p** operand, since both **libQ.a** and **libp.a** reside in the same directory.

   Multiple –**L** operands can be used when library name collisions occur. Building on the previous example, suppose that the user wants to use a new **libp.a**, in **/a/a/a**, but still wants *f*( ) from **/a/b/c/libQ.a**:

   ```
   c99 –L /a/a/a –L /a/b/c main.o a.c –l Q b.c –l p
   ```

   In this example, the linker searches the –**L** options in the order specified, and finds **/a/a/a/libp.a** before **/a/b/c/libp.a** when resolving references for **b.c**. The order of the –**l** operands is still important, however.

4. The following example shows how an application can use a programming environment where the widths of the following types:

   blksize_t, **cc_t**, **mode_t**, **nfds_t**, **pid_t**, **ptrdiff_t**, **size_t**, **speed_t**, **ssize_t**, **suseconds_t**, **tcflag_t**, **useconds_t**, **wchar_t**, **wint_t**

   are no greater than the width of type **long**:

   ```
   # First choose one of the listed environments ...

   # ... if there are no additional constraints, the first one will do:
   CENV=$(getconf _POSIX_V6_WIDTH_RESTRICTED_ENVS | head -n l)

   # ... or, if an environment that supports large files is preferred,
   # look for names that contain "OFF64" or "OFFBIG". (This chooses
   # the last one in the list if none match):
   for CENV in $(getconf _POSIX_V6_WIDTH_RESTRICTED_ENVS)
   do
       case $CENV in
       *OFF64*|*OFFBIG*) break ;;
       esac
   done

   # The chosen environment name can now be used like this:

   c99 $(getconf ${CENV}_CFLAGS) -D _POSIX_C_SOURCE=200xxxL \
   $(getconf ${CENV}_LDFLAGS) foo.c -o foo \
   $(getconf ${CENV} _LIBS)
   ```

RATIONALE

The *c99* utility is based on the *c89* utility originally introduced in the ISO POSIX-2: 1993 standard.

Some of the changes from *c89* include the modification to the contents of the Standard Libraries section to account for new headers and options; for example, <**spawn.h**> added to the –**l rt** operand, and the –**l** trace operand added for the Tracing functions.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *getconf*, *make*, *nm*, *strip*, *umask*, the System Interfaces volume of IEEE Std 1003.1-200x, *sysconf*( ), the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers |

CHANGE HISTORY

First released in Issue 6. Included for alignment with the ISO/IEC 9899:1999 standard.

NAME

cal — print a calendar

SYNOPSIS

XSI
```
cal [[month] year ]
```

DESCRIPTION

The *cal* utility shall write a calendar to standard output using the Julian calendar for dates from January 1, 1 through September 2, 1752 and the Gregorian calendar for dates from September 14, 1752 through December 31, 9999 as though the Gregorian calendar had been adopted on September 14, 1752.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

month	Specify the month to be displayed, represented as a decimal integer from 1 (January) to 12 (December). The default shall be the current month.
year	Specify the year for which the calendar is displayed, represented as a decimal integer from 1 to 9999. The default shall be the current year.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cal*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.
LC_TIME	Determine the format and contents of the calendar.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
TZ	Determine the timezone used to calculate the value of the current month.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be used to display the calendar, in an unspecified format.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Note that:

```
cal 83
```

refers to A.D. 83, not 1983.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

A future version of IEEE Std 1003.1-200x may support locale-specific recognition of the date of adoption of the Gregorian calendar. |

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 6

The DESCRIPTION is updated to allow for traditional behavior for years before the adoption of the Gregorian calendar.

NAME

cat — concatenate and print files

SYNOPSIS

```
cat [−u][file ...]
```

DESCRIPTION

The *cat* utility shall read files in sequence and shall write their contents to the standard output in the same sequence.

OPTIONS

The *cat* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−u	Write bytes from the input file to the standard output without delay as each is read.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If no *file* operands are specified, the standard input shall be used. If a *file* is '−', the *cat* utility shall read from the standard input at that point in the sequence. The *cat* utility shall not close and reopen standard input when it is referenced in this way, but shall accept multiple occurrences of '−' as a *file* operand.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is '−'. See the INPUT FILES section.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cat*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall contain the sequence of bytes read from the input files. Nothing else shall be written to the standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were output successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The −**u** option has value in prototyping non-blocking reads from FIFOs. The intent is to support the following sequence:

```
mkfifo foo
cat −u foo > /dev/tty13 &
cat −u > foo
```

It is unspecified whether standard output is or is not buffered in the default case. This is sometimes of interest when standard output is associated with a terminal, since buffering may delay the output. The presence of the −**u** option guarantees that unbuffered I/O is available. It is implementation-defined whether the *cat* utility buffers output if the −**u** option is not specified. Traditionally, the −**u** option is implemented using the equivalent of the *setvbuf*() function defined in the System Interfaces volume of IEEE Std 1003.1-200x.

EXAMPLES

The following command:

```
cat myfile
```

writes the contents of the file **myfile** to standard output.

The following command:

```
cat doc1 doc2 > doc.all
```

concatenates the files **doc1** and **doc2** and writes the result to **doc.all**.

Because of the shell language mechanism used to perform output redirection, a command such as this:

```
cat doc doc.end > doc
```

causes the original data in **doc** to be lost.

The command:

```
cat start − middle − end > file
```

when standard input is a terminal, gets two arbitrary pieces of input from the terminal with a single invocation of *cat*. Note, however, that if standard input is a regular file, this would be equivalent to the command:

```
cat start − middle /dev/null end > file
```

because the entire contents of the file would be consumed by *cat* the first time ’−’ was used as a *file* operand and an end-of-file condition would be detected immediately when ’−’ was referenced the second time.

RATIONALE

Historical versions of the *cat* utility include the options −**e**, −**t**, and −**v**, which permit the ends of lines, <tab>s, and invisible characters, respectively, to be rendered visible in the output. The standard developers omitted these options because they provide too fine a degree of control over what is made visible, and similar output can be obtained using a command such as:

```
sed −n −e ’s/$/$/’ −e l pathname
```

The −**s** option was omitted because it corresponds to different functions in BSD and System V-based systems. The BSD −**s** option to squeeze blank lines can be accomplished by the shell script shown in following example:

```
sed −n ’
# Write non-empty lines.
/./   {
      p
      d
      }
# Write a single empty line, then look for more empty lines.
/^$/  p
# Get next line, discard the held <newline> (empty line),
# and look for more empty lines.
:Empty
/^$/  {
      N
      s/.//
      b Empty
      }
# Write the non-empty line before going back to search
# for the first in a set of empty lines.
      p
’
```

The System V −**s** option to silence error messages can be accomplished by redirecting the standard error. Note that the BSD documentation for *cat* uses the term ‘‘blank line’’ to mean the same as the POSIX ‘‘empty line’’: a line consisting only of a <newline>.

The BSD −**n** option was omitted because similar functionality can be obtained from the −**n** option of the *pr* utility.

FUTURE DIRECTIONS

None.

SEE ALSO

more

CHANGE HISTORY

First released in Issue 2.

NAME

cd — change the working directory

SYNOPSIS

```
cd [−L] [−P] [directory]

cd −
```

DESCRIPTION

The *cd* utility shall change the working directory of the current shell execution environment (see Section 2.12 (on page 2263)) by executing the following steps in sequence. (In the following steps, the symbol **curpath** represents an intermediate value used to simplify the description of the algorithm used by *cd*. There is no requirement that **curpath** be made visible to the application.)

1. If no *directory* operand is given and the *HOME* environment variable is empty or undefined, the default behavior is implementation-defined and no further steps shall be taken.

2. If no *directory* operand is given and the *HOME* environment variable is set to a non-empty value, the *cd* utility shall behave as if the directory named in the *HOME* environment variable was specified as the *directory* operand.

3. If the *directory* operand begins with a slash character, set **curpath** to the operand and proceed to step 7.

4. If the first component of the *directory* operand is dot or dot-dot, proceed to step 6.

5. Starting with the first pathname in the colon-separated pathnames of *CDPATH* (see the ENVIRONMENT VARIABLES section) if the pathname is non-null, test if the concatenation of that pathname, a slash character, and the *directory* operand names a directory. If the pathname is null, test if the concatenation of dot, a slash character, and the operand names a directory. In either case, if the resulting string names an existing directory, set **curpath** to that string and proceed to step 7. Otherwise, repeat this step with the next pathname in *CDPATH* until all pathnames have been tested.

6. Set **curpath** to the string formed by the concatenation of the value of *PWD* a slash character, and the operand.

7. If the −**P** option is in effect, the *cd* utility shall perform actions equivalent to the *chdir*( ) function, called with **curpath** as the *path* argument. If these actions succeed, the *PWD* environment variable shall be set to an absolute pathname for the current working directory and shall not contain filename components that, in the context of pathname resolution, refer to a file of type symbolic link. If there is insufficient permission on the new directory, or on any parent of that directory, to determine the current working directory, the value of the *PWD* environment variable is unspecified. If the actions equivalent to *chdir*( ) fail for any reason, the *cd* utility shall display an appropriate error message and not alter the *PWD* environment variable. Whether the actions equivalent to *chdir*( ) succeed or fail, no further steps shall be taken.

8. The **curpath** value shall then be converted to canonical form as follows, considering each component from beginning to end, in sequence:

    a. Dot components and any slashes that separate them from the next component shall be deleted.

    b. For each dot-dot component, if there is a preceding component and it is neither root nor dot-dot, the preceding component, all slashes separating the preceding component from dot-dot, dot-dot, and all slashes separating dot-dot from the following component shall be deleted.

c. An implementation may further simplify **curpath** by removing any trailing slash characters that are not also leading slashes, replacing multiple non-leading consecutive slashes with a single slash, and replacing three or or more leading slashes with a single slash. If, as a result of this canonicalization, the **curpath** variable is null, no further steps shall be taken.

9. The *cd* utility shall then perform actions equivalent to the *chdir*( ) function called with **curpath** as the *path* argument. If these actions failed for any reason, the *cd* utility shall display an appropriate error message and no further steps shall be taken. The *PWD* environment variable shall be set to **curpath**.

If, during the execution of the above steps, the *PWD* environment variable is changed, the *OLDPWD* environment variable shall also be changed to the value of the old working directory (that is the current working directory immediately prior to the call to *cd*).

OPTIONS

The *cd* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

–L	Handle the operand dot-dot logically; symbolic link components shall not be resolved before dot-dot components are processed (see steps 5. and 6. in the DESCRIPTION).
–P	Handle the operand dot-dot physically; symbolic link components shall be resolved before dot-dot components are processed (see step 4. in the DESCRIPTION).

If both –**L** and –**P** options are specified, the last of these options shall be used and all others ignored. If neither –**L** nor –**P** is specified, the operand shall be handled dot-dot logically; see the DESCRIPTION.

OPERANDS

The following operands shall be supported:

directory An absolute or relative pathname of the directory that shall become the new working directory. The interpretation of a relative pathname by *cd* depends on the –**L** option and the *CDPATH* and *PWD* environment variables. If *directory* is an empty string, the results are unspecified.

– When a hyphen is used as the operand, this shall be equivalent to the command: |

```
cd "$OLDPWD" && pwd
```

which changes to the previous working directory and then writes its name.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cd*:

CDPATH A colon-separated list of pathnames that refer to directories. The *cd* utility shall use this list in its attempt to change the directory, as described in the DESCRIPTION. An empty string in place of a directory pathname represents the current directory.

	If *CDPATH* is not set, it shall be treated as if it were an empty string.
HOME	The name of the directory, used when no *directory* operand is specified.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
OLDPWD	A pathname of the previous working directory, used by *cd* –.
PWD	This variable shall be set as specified in the DESCRIPTION. If an application sets or unsets the value of *PWD*, the behavior of *cd* is unspecified.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If a non-empty directory name from *CDPATH* is used, or if *cd* – is used, an absolute pathname of the new working directory shall be written to the standard output as follows:

```
"%s\n", <new directory>
```

Otherwise, there shall be no output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The directory was successfully changed.
>0	An error occurred.

CONSEQUENCES OF ERRORS

The working directory shall remain unchanged.

APPLICATION USAGE

Since *cd* affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(cd /tmp)
nohup cd
find . −exec cd {} \;
```

it does not affect the working directory of the caller's environment.

The user must have execute (search) permission in *directory* in order to change to it.

EXAMPLES

None.

RATIONALE

The use of the *CDPATH* was introduced in the System V shell. Its use is analogous to the use of the *PATH* variable in the shell. The BSD C shell used a shell parameter *cdpath* for this purpose.

A common extension when *HOME* is undefined is to get the login directory from the user database for the invoking user. This does not occur on System V implementations.

Some historical shells, such as the KornShell, took special actions when the directory name contained a dot-dot component, selecting the logical parent of the directory, rather than the actual parent directory; that is, it moved up one level toward the `'/'` in the pathname, remembering what the user typed, rather than performing the equivalent of:

```
chdir("..");
```

In such a shell, the following commands would not necessarily produce equivalent output for all directories:

```
cd .. && ls      ls ..
```

This behavior is not permitted by default because it is not consistent with the definition of dot-dot in most historical practice; that is, while this behavior has been optionally available in the KornShell, other shells have historically not supported this functionality. The logical pathname is stored in the *PWD* environment variable when the *cd* utility completes and this value is used to construct the next directory name if *cd* is invoked with the −**L** option.

FUTURE DIRECTIONS

None.

SEE ALSO

pwd, the System Interfaces volume of IEEE Std 1003.1-200x, *chdir*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *cd* −, *PWD*, and *OLDPWD* are added.

The −**L** and −**P** options are added to align with the IEEE P1003.2b draft standard. This also includes the introduction of a new description to include the effect of these options.

NAME

cflow — generate a C-language flowgraph (**DEVELOPMENT**)

SYNOPSIS

XSI

```
cflow [−r][−d num][−D name[=def]] ... [−i incl][−I dir] ...
    [−U dir] ... file ...
```

DESCRIPTION

The *cflow* utility shall analyze a collection of object files or assembler, C-language, *lex* or *yacc* source files, and attempt to build a graph, written to standard output, charting the external references.

OPTIONS

The *cflow* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the order of the **−D**, **−I**, and **−U** options (which are identical to their interpretation by *c99*) is significant.

The following options shall be supported:

−d *num*	Indicate the depth at which the flowgraph is cut off. The application shall ensure that the argument *num* is a decimal integer. By default this is a very large number (typically greater than 32 000). Attempts to set the cut-off depth to a non-positive integer shall be ignored.
−i *incl*	Increase the number of included symbols. The *incl* option-argument is one of the following characters: *x* Include external and static data symbols. The default shall be to include only functions in the flowgraph. _ (Underscore) Include names that begin with an underscore. The default shall be to exclude these functions (and data if **−i x** is used).
−r	Reverse the caller:callee relationship, producing an inverted listing showing the callers of each function. The listing shall also be sorted in lexicographical order by callee.

OPERANDS

The following operand is supported:

file	The pathname of a file for which a graph is to be generated. Filenames suffixed by **.l** shall shall be taken to be *lex* input, **.y** as *yacc* input, **.c** as *c99* input, and **.i** as the output of *c99* **−E**. Such files shall be processed as appropriate, determined by their suffix. Files suffixed in **.s** (conventionally assembler source) may have more limited information extracted from them.

STDIN

Not used.

INPUT FILES

The input files shall be object files or assembler, C-language, *lex* or *yacc* source files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cflow*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2,

	Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the ordering of the output when the **–r** option is used.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The flowgraph written to standard output shall be formatted as follows:

`"%d %s:%s\n", <reference number>, <global>, <definition>`

Each line of output begins with a reference (that is, line) number, followed by indentation of at least one column position per level. This is followed by the name of the global, a colon, and its definition. Normally globals are only functions not defined as an external or beginning with an underscore; see the OPTIONS section for the **–i** inclusion option. For information extracted from C-language source, the definition consists of an abstract type declaration (for example, **char ***) and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the filename and location counter under which the symbol appeared (for example, *text*).

Once a definition of a name has been written, subsequent references to that name contain only the reference number of the line where the definition can be found. For undefined references, only "<>" shall be written.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Files produced by *lex* and *yacc* cause the reordering of line number declarations, and this can confuse *cflow*. To obtain proper results, the input of *yacc* or *lex* must be directed to *cflow*.

EXAMPLES

Given the following in **file.c**:

```
int i;
int f();
int g();
int h();
int
main()
{
    f();
    g();
    f();
}
int
f()
{
    i = h();
}
```

The command:

```
cflow −i x file.c
```

produces the output:

```
1 main: int(), <file.c 6>
2    f: int(), <file.c 13>
3        h: <>
4        i: int, <file.c 1>
5    g: <>
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *lex*, *yacc*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

chgrp — change the file group ownership

SYNOPSIS

```
chgrp –hR group file ...

chgrp –R [–H | –L | –P ] group file ...
```

DESCRIPTION

The *chgrp* utility shall set the group ID of the file named by each *file* operand to the group ID specified by the *group* operand.

For each *file* operand, or, if the **–R** option is used, each file encountered while walking the directory trees specified by the *file* operands, the *chgrp* utility shall perform actions equivalent to the *chown*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called with the following arguments:

- The *file* operand shall be used as the *path* argument.
- The user ID of the file shall be used as the *owner* argument.
- The specified group ID shall be used as the *group* argument.

Unless *chgrp* is invoked by a process with appropriate privileges, the set-user-ID and set-group-ID bits of a regular file shall be cleared upon successful completion; the set-user-ID and set-group-ID bits of other file types may be cleared.

OPTIONS

The *chgrp* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

–h	If the system supports group IDs for symbolic links, for each *file* operand that names a file of type symbolic link, *chgrp* shall attempt to set the group ID of the symbolic link instead of the file referenced by the symbolic link. If the system does not support group IDs for symbolic links, for each *file* operand that names a file of type symbolic link, *chgrp* shall do nothing more with the current file and shall go on to any remaining files.
–H	If the **–R** option is specified and a symbolic link referencing a file of type directory is specified on the command line, *chgrp* shall change the group of the directory referenced by the symbolic link and all files in the file hierarchy below it.
–L	If the **–R** option is specified and a symbolic link referencing a file of type directory is specified on the command line or encountered during the traversal of a file hierarchy, *chgrp* shall change the group of the directory referenced by the symbolic link and all files in the file hierarchy below it.
–P	If the **–R** option is specified and a symbolic link is specified on the command line or encountered during the traversal of a file hierarchy, *chgrp* shall change the group ID of the symbolic link if the system supports this operation. The *chgrp* utility shall not follow the symbolic link to any other part of the file hierarchy.
–R	Recursively change file group IDs. For each *file* operand that names a directory, *chgrp* shall change the group of the directory and all files in the file hierarchy below it. Unless a **–H**, **–L**, or **–P** option is specified, it is unspecified which of these options will be used as the default.

Specifying more than one of the mutually-exclusive options **–H**, **–L**, and **–P** shall not be considered an error. The last option specified shall determine the behavior of the utility.

OPERANDS

The following operands shall be supported:

group	A group name from the group database or a numeric group ID. Either specifies a group ID to be given to each file named by one of the *file* operands. If a numeric *group* operand exists in the group database as a group name, the group ID number associated with that group name is used as the group ID.
file	A pathname of a file whose group ID is to be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *chgrp*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The utility executed successfully and all requested changes were made.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Only the owner of a file or the user with appropriate privileges may change the owner or group of a file.

Some implementations restrict the use of *chgrp* to a user with appropriate privileges when the *group* specified is not the effective group ID or one of the supplementary group IDs of the calling process. |

EXAMPLES

None.

RATIONALE

The System V and BSD versions use different exit status codes. Some implementations used the exit status as a count of the number of errors that occurred; this practice is unworkable since it can overflow the range of valid exit status values. The standard developers chose to mask these by specifying only 0 and >0 as exit values.

The functionality of *chgrp* is described substantially through references to *chown*( ). In this way, there is no duplication of effort required for describing the interactions of permissions, multiple groups, and so on.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, *chown*, the System Interfaces volume of IEEE Std 1003.1-200x, *chown*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

New options −**H**, −**L**, and −**P** are added to align with the IEEE P1003.2b draft standard. These options affect the processing of symbolic links.

IEEE PASC Interpretation 1003.2 #172 is applied, changing the CONSEQUENCES OF ERRORS section to ‘‘Default.’’.

NAME

chmod — change the file modes

SYNOPSIS

```
chmod [−R] mode file ...
```

DESCRIPTION

The *chmod* utility shall change any or all of the file mode bits of the file named by each *file* operand in the way specified by the *mode* operand.

It is implementation-defined whether and how the *chmod* utility affects any alternate or additional file access control mechanism (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.4, File Access Permissions) being used for the specified file.

Only a process whose effective user ID matches the user ID of the file, or a process with the appropriate privileges, shall be permitted to change the file mode bits of a file.

OPTIONS

The *chmod* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−R	Recursively change file mode bits. For each *file* operand that names a directory, *chmod* shall change the file mode bits of the directory and all files in the file hierarchy below it.

OPERANDS

The following operands shall be supported:

mode	Represents the change to be made to the file mode bits of each file named by one of the *file* operands; see the EXTENDED DESCRIPTION section.
file	A pathname of a file whose file mode bits shall be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *chmod*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The *mode* operand shall be either a *symbolic_mode* expression or a non-negative octal integer. The *symbolic_mode* form is described by the grammar later in this section.

Each **clause** shall specify an operation to be performed on the current file mode bits of each *file*. The operations shall be performed on each *file* in the order in which the **clause**s are specified.

The **who** symbols **u**, **g**, and **o** shall specify the *user*, *group*, and *other* parts of the file mode bits, respectively. A **who** consisting of the symbol **a** shall be equivalent to **ugo**.

The **perm** symbols **r**, **w**, and **x** represent the *read*, *write*, and *execute*/*search* portions of file mode bits, respectively. The **perm** symbol **s** shall represent the *set-user-ID-on-execution* (when **who** contains or implies **u**) and *set-group-ID-on-execution* (when **who** contains or implies **g**) bits.

The **perm** symbol **X** shall represent the execute/search portion of the file mode bits if the file is a directory or if the current (unmodified) file mode bits have at least one of the execute bits (S_IXUSR, S_IXGRP, or S_IXOTH) set. It shall be ignored if the file is not a directory and none of the execute bits are set in the current file mode bits.

The **permcopy** symbols **u**, **g**, and **o** shall represent the current permissions associated with the user, group, and other parts of the file mode bits, respectively. For the remainder of this section, **perm** refers to the non-terminals **perm** and **permcopy** in the grammar.

If multiple **actionlist**s are grouped with a single **wholist** in the grammar, each **actionlist** shall be applied in the order specified with that **wholist**. The *op* symbols shall represent the operation performed, as follows:

\+ If **perm** is not specified, the '+' operation shall not change the file mode bits.

If **who** is not specified, the file mode bits represented by **perm** for the owner, group, and other permissions, except for those with corresponding bits in the file mode creation mask of the invoking process, shall be set.

Otherwise, the file mode bits represented by the specified **who** and **perm** values shall be set.

– If **perm** is not specified, the '–' operation shall not change the file mode bits.

If **who** is not specified, the file mode bits represented by **perm** for the owner, group, and other permissions, except for those with corresponding bits in the file mode creation mask of the invoking process, shall be cleared.

Otherwise, the file mode bits represented by the specified **who** and **perm** values shall be cleared.

= Clear the file mode bits specified by the **who** value, or, if no **who** value is specified, all of the file mode bits specified in this volume of IEEE Std 1003.1-200x.

If **perm** is not specified, the '=' operation shall make no further modifications to the file mode bits.

If **who** is not specified, the file mode bits represented by **perm** for the owner, group, and other permissions, except for those with corresponding bits in the file mode creation mask of the invoking process, shall be set.

Otherwise, the file mode bits represented by the specified **who** and **perm** values shall be set.

When using the symbolic mode form on a regular file, it is implementation-defined whether or not:

- Requests to set the set-user-ID-on-execution or set-group-ID-on-execution bit when all execute bits are currently clear and none are being set are ignored.
- Requests to clear all execute bits also clear the set-user-ID-on-execution and set-group-ID-on-execution bits.
- Requests to clear the set-user-ID-on-execution or set-group-ID-on-execution bits when all execute bits are currently clear are ignored. However, if the command *ls* **–l** *file* writes an *s* in the position indicating that the set-user-ID-on-execution or set-group-ID-on-execution is set, the commands *chmod* **u–s** *file* or *chmod* **g–s** *file*, respectively, shall not be ignored.

When using the symbolic mode form on other file types, it is implementation-defined whether or not requests to set or clear the set-user-ID-on-execution or set-group-ID-on-execution bits are honored.

If the **who** symbol **o** is used in conjunction with the **perm** symbol **s** with no other **who** symbols being specified, the set-user-ID-on-execution and set-group-ID-on-execution bits shall not be modified. It shall not be an error to specify the **who** symbol **o** in conjunction with the **perm** symbol **s**.

For an octal integer *mode* operand, the file mode bits shall be set absolutely.

For each bit set in the octal number, the corresponding file permission bit shown in the following table shall be set; all other file permission bits shall be cleared. For regular files, for each bit set in the octal number corresponding to the set-user-ID-on-execution or the set-group-ID-on-execution, bits shown in the following table shall be set; if these bits are not set in the octal number, they are cleared. For other file types, it is implementation-defined whether or not requests to set or clear the set-user-ID-on-execution or set-group-ID-on-execution bits are honored.

Octal	Mode Bit	Octal	Mode Bit	Octal	Mode Bit	Octal	Mode Bit
4000	S_ISUID	**0400**	S_IRUSR	**0040**	S_IRGRP	**0004**	S_IROTH
2000	S_ISGID	**0200**	S_IWUSR	**0020**	S_IWGRP	**0002**	S_IWOTH
		0100	S_IXUSR	**0010**	S_IXGRP	**0001**	S_IXOTH

When bits are set in the octal number other than those listed in the table above, the behavior is unspecified.

Grammar for chmod

The grammar and lexical conventions in this section describe the syntax for the *symbolic_mode* operand. The general conventions for this style of grammar are described in Section 1.10 (on page 2220). A valid *symbolic_mode* can be represented as the non-terminal symbol *symbolic_mode* in the grammar. This formal syntax shall take precedence over the preceding text syntax description.

The lexical processing is based entirely on single characters. Implementations need not allow blank characters within the single argument being processed.

```
%start    symbolic_mode
%%

symbolic_mode    : section
                 | symbolic_mode ',' clause
                 ;

clause           : actionlist
                 | wholist actionlist
                 ;

wholist          : who
                 | wholist who
                 ;

who              : 'u' | 'g' | 'o' | 'a'
                 ;

actionlist       : action
                 | actionlist action
                 ;

action           : op
                 | op permlist
                 | op permcopy
                 ;

permcopy         : 'u' | 'g' | 'o'
                 ;

op               : '+' | '-' | '='
                 ;

permlist         : perm
                 | perm permlist
                 ;

perm             : 'r' | 'w' | 'x' | 'X' | 's'
                 ;
```

EXIT STATUS

The following exit values shall be returned:

0	The utility executed successfully and all requested changes were made.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Some implementations of the *chmod* utility change the mode of a directory before the files in the directory when performing a recursive (−**R** option) change; others change the directory mode after the files in the directory. If an application tries to remove read or search permission for a file hierarchy, the removal attempt fails if the directory is changed first; on the other hand, trying to re-enable permissions to a restricted hierarchy fails if directories are changed last. Users should not try to make a hierarchy inaccessible to themselves.

Some implementations of *chmod* never used the process' *umask* when changing modes; systems conformant with this volume of IEEE Std 1003.1-200x do so when **who** is not specified. Note the difference between:

```
chmod a-w file
```

which removes all write permissions, and:

```
chmod -- -w file
```

which removes write permissions that would be allowed if **file** was created with the same *umask*.

Conforming applications should never assume that they know how the set-user-ID and set-group-ID bits on directories are interpreted.

EXAMPLES

Mode	Results
a+=	Equivalent to *a*+,*a*=; clears all file mode bits.
go+−w	Equivalent to *go*+,*go*−*w*; clears group and other write bits.
g=*o*−*w*	Equivalent to *g*=*o*,*g*−*w*; sets group bit to match other bits and then clears group write bit.
g−*r*+*w*	Equivalent to *g*−*r*,*g*+*w*; clears group read bit and sets group write bit.
=*g*	Sets owner bits to match group bits and sets other bits to match group bits.

RATIONALE

The functionality of *chmod* is described substantially through references to concepts defined in the System Interfaces volume of IEEE Std 1003.1-200x. In this way, there is less duplication of effort required for describing the interactions of permissions. However, the behavior of this utility is not described in terms of the *chmod*( ) function from the System Interfaces volume of IEEE Std 1003.1-200x because that specification requires certain side effects upon alternate file access control mechanisms that might not be appropriate, depending on the implementation.

Implementations that support mandatory file and record locking as specified by the 1984 /usr/group standard historically used the combination of set-group-ID bit set and group execute bit clear to indicate mandatory locking. This condition is usually set or cleared with the symbolic mode **perm** symbol **l** instead of the **perm** symbols **s** and **x** so that the mandatory locking mode is not changed without explicit indication that that was what the user intended. Therefore, the details on how the implementation treats these conditions must be defined in the documentation. This volume of IEEE Std 1003.1-200x does not require mandatory locking (nor does the System Interfaces volume of IEEE Std 1003.1-200x), but does allow it as an extension. However, this volume of IEEE Std 1003.1-200x does require that the *ls* and *chmod* utilities work consistently in this area. If *ls* **–l** *file* indicates that the set-group-ID bit is set, *chmod* **g–s** *file* must clear it (assuming appropriate privileges exist to change modes).

The System V and BSD versions use different exit status codes. Some implementations used the exit status as a count of the number of errors that occurred; this practice is unworkable since it can overflow the range of valid exit status values. This problem is avoided here by specifying only 0 and >0 as exit values.

The System Interfaces volume of IEEE Std 1003.1-200x indicates that implementation-defined restrictions may cause the S_ISUID and S_ISGID bits to be ignored. This volume of IEEE Std 1003.1-200x allows the *chmod* utility to choose to modify these bits before calling *chmod*( ) (or some function providing equivalent capabilities) for non-regular files. Among other things, this allows implementations that use the set-user-ID and set-group-ID bits on directories to enable extended features to handle these extensions in an intelligent manner.

The **X perm** symbol was adopted from BSD-based systems because it provides commonly desired functionality when doing recursive (**–R** option) modifications. Similar functionality is not provided by the *find* utility. Historical BSD versions of *chmod*, however, only supported **X** with *op*+; it has been extended in this volume of IEEE Std 1003.1-200x because it is also useful with *op*=. (It has also been added for *op*– even though it duplicates **x**, in this case, because it is intuitive and easier to explain.)

The grammar was extended with the *permcopy* non-terminal to allow historical-practice forms of symbolic modes like **o**=**u** –**g** (that is, set the ‘‘other’’ permissions to the permissions of ‘‘owner’’ minus the permissions of ‘‘group’’).

FUTURE DIRECTIONS

None.

SEE ALSO

ls, *umask*, the System Interfaces volume of IEEE Std 1003.1-200x, *chmod*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- Octal modes have been kept and made mandatory despite being marked obsolescent in the previous version of this volume of IEEE Std 1003.1-200x.

IEEE PASC Interpretation 1003.2 #172 is applied, changing the CONSEQUENCES OF ERRORS section to ‘‘Default.’’.

NAME

chown — change the file ownership

SYNOPSIS

`chown −hR owner[:group] file ...`

`chown −R [−H | −L | −P ] owner[:group] file ...`

DESCRIPTION

The *chown* utility shall set the user ID of the file named by each *file* operand to the user ID specified by the *owner* operand.

For each *file* operand, or, if the **−R** option is used, each file encountered while walking the directory trees specified by the *file* operands, the *chown* utility shall perform actions equivalent to the *chown*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called with the following arguments:

1. The *file* operand shall be used as the *path* argument.
2. The user ID indicated by the *owner* portion of the first operand shall be used as the *owner* argument.
3. If the *group* portion of the first operand is given, the group ID indicated by it shall be used as the *group* argument; otherwise, the group ID of the file shall be used as the *group* argument.

Unless *chown* is invoked by a process with appropriate privileges, the set-user-ID and set-group-ID bits of a regular file shall be cleared upon successful completion; the set-user-ID and set-group-ID bits of other file types may be cleared.

OPTIONS

The *chown* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−h If the system supports user IDs for symbolic links, for each *file* operand that names a file of type symbolic link, *chown* shall attempt to set the user ID of the symbolic link. If the system supports group IDs for symbolic links, and a group ID was specified, for each *file* operand that names a file of type symbolic link, *chown* shall attempt to set the group ID of the symbolic link. If the system does not support user or group IDs for symbolic links, for each *file* operand that names a file of type symbolic link, *chown* shall do nothing more with the current file and shall go on to any remaining files.

−H If the **−R** option is specified and a symbolic link referencing a file of type directory is specified on the command line, *chown* shall change the user ID (and group ID, if specified) of the directory referenced by the symbolic link and all files in the file hierarchy below it.

−L If the **−R** option is specified and a symbolic link referencing a file of type directory is specified on the command line or encountered during the traversal of a file hierarchy, *chown* shall change the user ID (and group ID, if specified) of the directory referenced by the symbolic link and all files in the file hierarchy below it.

−P If the **−R** option is specified and a symbolic link is specified on the command line or encountered during the traversal of a file hierarchy, *chown* shall change the owner ID (and group ID, if specified) of the symbolic link if the system supports this operation. The *chown* utility shall not follow the symbolic link to any other part of the file hierarchy.

–R	Recursively change file user and group IDs. For each *file* operand that names a directory, *chown* shall change the user ID (and group ID, if specified) of the directory and all files in the file hierarchy below it. Unless a **–H**, **–L**, or **–P** option is specified, it is unspecified which of these options will be used as the default.

Specifying more than one of the mutually-exclusive options **–H**, **–L**, and **–P** shall not be considered an error. The last option specified shall determine the behavior of the utility.

OPERANDS

The following operands shall be supported:

owner[:*group*]	A user ID and optional group ID to be assigned to *file*. The *owner* portion of this operand shall be a user name from the user database or a numeric user ID. Either specifies a user ID which shall be given to each file named by one of the *file* operands. If a numeric *owner* operand exists in the user database as a user name, the user ID number associated with that user name shall be used as the user ID. Similarly, if the *group* portion of this operand is present, it shall be a group name from the group database or a numeric group ID. Either specifies a group ID which shall be given to each file. If a numeric group operand exists in the group database as a group name, the group ID number associated with that group name shall be used as the group ID.
file	A pathname of a file whose user ID is to be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *chown*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 The utility executed successfully and all requested changes were made.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Only the owner of a file or the user with appropriate privileges may change the owner or group of a file.

Some implementations restrict the use of *chown* to a user with appropriate privileges.

EXAMPLES

None.

RATIONALE

The System V and BSD versions use different exit status codes. Some implementations used the exit status as a count of the number of errors that occurred; this practice is unworkable since it can overflow the range of valid exit status values. These are masked by specifying only 0 and >0 as exit values.

The functionality of *chown* is described substantially through references to functions in the System Interfaces volume of IEEE Std 1003.1-200x. In this way, there is no duplication of effort required for describing the interactions of permissions, multiple groups, and so on.

The 4.3 BSD method of specifying both owner and group was included in this volume of IEEE Std 1003.1-200x because:

- There are cases where the desired end condition could not be achieved using the *chgrp* and *chown* (that only changed the user ID) utilities. (If the current owner is not a member of the desired group and the desired owner is not a member of the current group, the *chown*( ) function could fail unless both owner and group are changed at the same time.)
- Even if they could be changed independently, in cases where both are being changed, there is a 100% performance penalty caused by being forced to invoke both utilities.

The BSD syntax *user*[.*group*] was changed to *user*[:*group*] in this volume of IEEE Std 1003.1-200x because the period is a valid character in login names (as specified by the Base Definitions volume of IEEE Std 1003.1-200x, login names consist of characters in the portable filename character set). The colon character was chosen as the replacement for the period character because it would never be allowed as a character in a user name or group name on historical implementations.

The −**R** option is considered by some observers as an undesirable departure from the historical UNIX system tools approach; since a tool, *find*, already exists to recurse over directories, there seemed to be no good reason to require other tools to have to duplicate that functionality. However, the –**R** option was deemed an important user convenience, is far more efficient than forking a separate process for each element of the directory hierarchy, and is in widespread historical use.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, *chgrp*, the System Interfaces volume of IEEE Std 1003.1-200x, *chown*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

New options –**h**, –**H**, –**L**, and –**P** are added to align with the IEEE P1003.2b draft standard. These options affect the processing of symbolic links.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

IEEE PASC Interpretation 1003.2 #172 is applied, changing the CONSEQUENCES OF ERRORS section is changed to ‘‘Default.’’.

NAME

cksum — write file checksums and sizes

SYNOPSIS

```
cksum [file ...]
```

DESCRIPTION

The *cksum* utility shall calculate and write to standard output a cyclic redundancy check (CRC) for each input file, and also write to standard output the number of octets in each file. The CRC used is based on the polynomial used for CRC error checking in the ISO/IEC 8802-3:1996 standard (Ethernet).

The encoding for the CRC checksum is defined by the generating polynomial:

$$G(x)=x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$$

Mathematically, the CRC value corresponding to a given file shall be defined by the following procedure:

1. The n bits to be evaluated are considered to be the coefficients of a mod 2 polynomial $M(x)$ of degree $n-1$. These n bits are the bits from the file, with the most significant bit being the most significant bit of the first octet of the file and the last bit being the least significant bit of the last octet, padded with zero bits (if necessary) to achieve an integral number of octets, followed by one or more octets representing the length of the file as a binary value, least significant octet first. The smallest number of octets capable of representing this integer shall be used.

2. $M(x)$ is multiplied by x^{32} (that is, shifted left 32 bits) and divided by $G(x)$ using mod 2 division, producing a remainder $R(x)$ of degree ≤ 31.

3. The coefficients of $R(x)$ are considered to be a 32-bit sequence.

4. The bit sequence is complemented and the result is the CRC.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

file	A pathname of a file to be checked. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cksum*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

For each file processed successfully, the *cksum* utility shall write in the following format:

```
"%u %d %s\n", <checksum>, <# of octets>, <pathname>
```

If no *file* operand was specified, the pathname and its leading <space> shall be omitted.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All files were processed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *cksum* utility is typically used to quickly compare a suspect file against a trusted version of the same, such as to ensure that files transmitted over noisy media arrive intact. However, this comparison cannot be considered cryptographically secure. The chances of a damaged file producing the same CRC as the original are small; deliberate deception is difficult, but probably not impossible.

Although input files to *cksum* can be any type, the results need not be what would be expected on character special device files or on file types not described by the System Interfaces volume of IEEE Std 1003.1-200x. Since this volume of IEEE Std 1003.1-200x does not specify the block size used when doing input, checksums of character special files need not process all of the data in those files.

The algorithm is expressed in terms of a bitstream divided into octets. If a file is transmitted between two systems and undergoes any data transformation (such as changing little-endian byte ordering to big-endian), identical CRC values cannot be expected. Implementations performing such transformations may extend *cksum* to handle such situations.

EXAMPLES

None.

RATIONALE

The following C-language program can be used as a model to describe the algorithm. It assumes that a **char** is one octet. It also assumes that the entire file is available for one pass through the function. This was done for simplicity in demonstrating the algorithm, rather than as an implementation model.

```
static unsigned long crctab[] = {
0x00000000,
0x04c11db7, 0x09823b6e, 0x0d4326d9, 0x130476dc, 0x17c56b6b,
0x1a864db2, 0x1e475005, 0x2608edb8, 0x22c9f00f, 0x2f8ad6d6,
0x2b4bcb61, 0x350c9b64, 0x31cd86d3, 0x3c8ea00a, 0x384fbdbd,
0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9, 0x5f15adac,
0x5bd4b01b, 0x569796c2, 0x52568b75, 0x6a1936c8, 0x6ed82b7f,
0x639b0da6, 0x675a1011, 0x791d4014, 0x7ddc5da3, 0x709f7b7a,
0x745e66cd, 0x9823b6e0, 0x9ce2ab57, 0x91a18d8e, 0x95609039,
0x8b27c03c, 0x8fe6dd8b, 0x82a5fb52, 0x8664e6e5, 0xbe2b5b58,
0xbaea46ef, 0xb7a96036, 0xb3687d81, 0xad2f2d84, 0xa9ee3033,
0xa4ad16ea, 0xa06c0b5d, 0xd4326d90, 0xd0f37027, 0xddb056fe,
0xd9714b49, 0xc7361b4c, 0xc3f706fb, 0xceb42022, 0xca753d95,
0xf23a8028, 0xf6fb9d9f, 0xfbb8bb46, 0xff79a6f1, 0xe13ef6f4,
0xe5ffeb43, 0xe8bccd9a, 0xec7dd02d, 0x34867077, 0x30476dc0,
0x3d044b19, 0x39c556ae, 0x278206ab, 0x23431b1c, 0x2e003dc5,
0x2ac12072, 0x128e9dcf, 0x164f8078, 0x1b0ca6a1, 0x1fcdbb16,
0x018aeb13, 0x054bf6a4, 0x0808d07d, 0x0cc9cdca, 0x7897ab07,
0x7c56b6b0, 0x71159069, 0x75d48dde, 0x6b93dddb, 0x6f52c06c,
0x6211e6b5, 0x66d0fb02, 0x5e9f46bf, 0x5a5e5b08, 0x571d7dd1,
0x53dc6066, 0x4d9b3063, 0x495a2dd4, 0x44190b0d, 0x40d816ba,
0xaca5c697, 0xa864db20, 0xa527fdf9, 0xa1e6e04e, 0xbfa1b04b,
0xbb60adfc, 0xb6238b25, 0xb2e29692, 0x8aad2b2f, 0x8e6c3698,
0x832f1041, 0x87ee0df6, 0x99a95df3, 0x9d684044, 0x902b669d,
0x94ea7b2a, 0xe0b41de7, 0xe4750050, 0xe9362689, 0xedf73b3e,
0xf3b06b3b, 0xf771768c, 0xfa325055, 0xfef34de2, 0xc6bcf05f,
0xc27dede8, 0xcf3ecb31, 0xcbffd686, 0xd5b88683, 0xd1799b34,
0xdc3abded, 0xd8fba05a, 0x690ce0ee, 0x6dcdfd59, 0x608edb80,
0x644fc637, 0x7a089632, 0x7ec98b85, 0x738aad5c, 0x774bb0eb,
0x4f040d56, 0x4bc510e1, 0x46863638, 0x42472b8f, 0x5c007b8a,
0x58c1663d, 0x558240e4, 0x51435d53, 0x251d3b9e, 0x21dc2629,
0x2c9f00f0, 0x285e1d47, 0x36194d42, 0x32d850f5, 0x3f9b762c,
0x3b5a6b9b, 0x0315d626, 0x07d4cb91, 0x0a97ed48, 0x0e56f0ff,
0x1011a0fa, 0x14d0bd4d, 0x19939b94, 0x1d528623, 0xf12f560e,
0xf5ee4bb9, 0xf8ad6d60, 0xfc6c70d7, 0xe22b20d2, 0xe6ea3d65,
0xeba91bbc, 0xef68060b, 0xd727bbb6, 0xd3e6a601, 0xdea580d8,
0xda649d6f, 0xc423cd6a, 0xc0e2d0dd, 0xcda1f604, 0xc960ebb3,
0xbd3e8d7e, 0xb9ff90c9, 0xb4bcb610, 0xb07daba7, 0xae3afba2,
0xaafbe615, 0xa7b8c0cc, 0xa379dd7b, 0x9b3660c6, 0x9ff77d71,
0x92b45ba8, 0x9675461f, 0x8832161a, 0x8cf30bad, 0x81b02d74,
0x857130c3, 0x5d8a9099, 0x594b8d2e, 0x5408abf7, 0x50c9b640,
0x4e8ee645, 0x4a4ffbf2, 0x470cdd2b, 0x43cdc09c, 0x7b827d21,
0x7f436096, 0x7200464f, 0x76c15bf8, 0x68860bfd, 0x6c47164a,
0x61043093, 0x65c52d24, 0x119b4be9, 0x155a565e, 0x18197087,
```

```
0x1cd86d30, 0x029f3d35, 0x065e2082, 0x0b1d065b, 0x0fdc1bec,
0x3793a651, 0x3352bbe6, 0x3e119d3f, 0x3ad08088, 0x2497d08d,
0x2056cd3a, 0x2d15ebe3, 0x29d4f654, 0xc5a92679, 0xc1683bce,
0xcc2b1d17, 0xc8ea00a0, 0xd6ad50a5, 0xd26c4d12, 0xdf2f6bcb,
0xdbee767c, 0xe3a1cbc1, 0xe760d676, 0xea23f0af, 0xeee2ed18,
0xf0a5bd1d, 0xf464a0aa, 0xf9278673, 0xfde69bc4, 0x89b8fd09,
0x8d79e0be, 0x803ac667, 0x84fbdbd0, 0x9abc8bd5, 0x9e7d9662,
0x933eb0bb, 0x97ffad0c, 0xafb010b1, 0xab710d06, 0xa6322bdf,
0xa2f33668, 0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4
};

unsigned long memcrc(const unsigned char *b, size_t n)
{
/*  Input arguments:
 *  const char*   b == byte sequence to checksum
 *  size_t        n == length of sequence
 */

    register unsigned   i, c, s = 0;

    for (i = n; i > 0; --i) {
        c = (unsigned)(*b++);
        s = (s << 8) ^ crctab[(s >> 24) ^ c];
    }

    /* Extend with the length of the string. */
    while (n != 0) {
        c = n & 0377;
        n >>= 8;
        s = (s << 8) ^ crctab[(s >> 24) ^ c];
    }

    return ~s;
}
```

The historical practice of writing the number of ‘‘blocks’’ has been changed to writing the number of octets, since the latter is not only more useful, but also since historical implementations have not been consistent in defining what a ‘‘block’’ meant. Octets are used instead of bytes because bytes can differ in size between systems.

The algorithm used was selected to increase the operational robustness of *cksum*. Neither the System V nor BSD *sum* algorithm was selected. Since each of these was different and each was the default behavior on those systems, no realistic compromise was available if either were selected—some set of historical applications would break. Therefore, the name was changed to *cksum*. Although the historical *sum* commands will probably continue to be provided for many years, programs designed for portability across systems should use the new name.

The algorithm selected is based on that used by the ISO / IEC 8802-3: 1996 standard (Ethernet) for the frame check sequence field. The algorithm used does not match the technical definition of a *checksum*; the term is used for historical reasons. The length of the file is included in the CRC calculation because this parallels inclusion of a length field by Ethernet in its CRC, but also because it guards against inadvertent collisions between files that begin with different series of zero octets. The chance that two different files produce identical CRCs is much greater when their lengths are not considered. Keeping the length and the checksum of the file itself separate would yield a slightly more robust algorithm, but historical usage has always been that a single number (the checksum as printed) represents the signature of the file. It was decided that historical usage was the more important consideration.

Early proposals contained modifications to the Ethernet algorithm that involved extracting table values whenever an intermediate result became zero. This was demonstrated to be less robust than the current method and mathematically difficult to describe or justify.

The calculation used is identical to that given in pseudo-code in the referenced Sarwate article. The pseudo-code rendition is:

```
X <– 0; Y <– 0;
for i <– m –1 step –1 until 0 do
    begin
    T <– X(1) ˆ A[i];
    X(1) <– X(0); X(0) <– Y(1); Y(1) <– Y(0); Y(0) <– 0;
    comment: f[T] and f’[T] denote the T-th words in the
        table f and f’ ;
    X <– X ˆ f[T]; Y <– Y ˆ f’[T];
    end
```

The pseudo-code is reproduced exactly as given; however, note that in the case of *cksum*, **A[i]** represents a byte of the file, the words **X** and **Y** are treated as a single 32-bit value, and the tables **f** and **f’** are a single table containing 32-bit values.

The referenced Sarwate article also discusses generating the table.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 4.

NAME

cmp — compare two files

SYNOPSIS

`cmp [ −l | −s ] file1 file2`

DESCRIPTION

The *cmp* utility shall compare two files. The *cmp* utility shall write no output if the files are the same. Under default options, if they differ, it shall write to standard output the byte and line number at which the first difference occurred. Bytes and lines shall be numbered beginning with 1.

OPTIONS

The *cmp* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−l	(Lowercase ell.) Write the byte number (decimal) and the differing bytes (octal) for each difference.
−s	Write nothing for differing files; return exit status only.

OPERANDS

The following operands shall be supported:

file1	A pathname of the first file to be compared. If *file1* is `'−'`, the standard input shall be used.
file2	A pathname of the second file to be compared. If *file2* is `'−'`, the standard input shall be used.

If both *file1* and *file2* refer to standard input or refer to the same FIFO special, block special, or character special file, the results are undefined.

STDIN

The standard input shall be used only if the *file1* or *file2* operand refers to standard input. See the INPUT FILES section.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cmp*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

In the POSIX locale, results of the comparison shall be written to standard output. When no options are used, the format shall be:

```
"%s %s differ: char %d, line %d\n", file1, file2,
    <byte number>, <line number>
```

When the **–l** option is used, the format shall be:

```
"%d %o %o\n", <byte number>, <differing byte>,
    <differing byte>
```

for each byte that differs. The first <*differing byte*> number is from *file1* while the second is from *file2*. In both cases, <*byte number*> shall be relative to the beginning of the file, beginning with 1.

No output shall be written to standard output when the **–s** option is used.

STDERR

The standard error shall be used only for diagnostic messages. If *file1* and *file2* are identical for the entire length of the shorter file, in the POSIX locale the following diagnostic message shall be written, unless the **–s** option is specified:

```
"cmp: EOF on %s%s\n", <name of shorter file>, <additional info>
```

The <*additional info*> field shall either be null or a string that starts with a <blank> and contains no <newline>s. Some implementations report on the number of lines in this case.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The files are identical.
1	The files are different; this includes the case where one file is identical to the first part of the other.
>1	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Although input files to *cmp* can be any type, the results might not be what would be expected on character special device files or on file types not described by the System Interfaces volume of IEEE Std 1003.1-200x. Since this volume of IEEE Std 1003.1-200x does not specify the block size used when doing input, comparisons of character special files need not compare all of the data in those files.

For files which are not text files, line numbers simply reflect the presence of a <newline>, without any implication that the file is organized into lines.

EXAMPLES

None.

RATIONALE

The global language in Section 1.11 (on page 2221) indicates that using two mutually-exclusive options together produces unspecified results. Some System V implementations consider the option usage:

```
cmp -l -s ...
```

to be an error. They also treat:

```
cmp -s -l ...
```

as if no options were specified. Both of these behaviors are considered bugs, but are allowed.

The word **char** in the standard output format comes from historical usage, even though it is actually a byte number. When *cmp* is supported in other locales, implementations are encouraged to use the word *byte* or its equivalent in another language. Users should not interpret this difference to indicate that the functionality of the utility changed between locales.

Some implementations report on the number of lines in the identical-but-shorter file case. This is allowed by the inclusion of the <*additional info*> fields in the output format. The restriction on having a leading <blank> and no <newline>s is to make parsing for the filename easier. It is recognized that some filenames containing white-space characters make parsing difficult anyway, but the restriction does aid programs used on systems where the names are predominantly well behaved.

FUTURE DIRECTIONS

None.

SEE ALSO

comm, *diff*

CHANGE HISTORY

First released in Issue 2.

NAME

comm — select or reject lines common to two files

SYNOPSIS

```
comm [−123] file1 file2
```

DESCRIPTION

The *comm* utility shall read *file1* and *file2*, which should be ordered in the current collating sequence, and produce three text columns as output: lines only in *file1*, lines only in *file2*, and lines in both files.

If the lines in both files are not ordered according to the collating sequence of the current locale, the results are unspecified.

OPTIONS

The *comm* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−1	Suppress the output column of lines unique to *file1*.
−2	Suppress the output column of lines unique to *file2*.
−3	Suppress the output column of lines duplicated in *file1* and *file2*.

OPERANDS

The following operands shall be supported:

file1	A pathname of the first file to be compared. If *file1* is `'−'`, the standard input shall be used.
file2	A pathname of the second file to be compared. If *file2* is `'−'`, the standard input shall be used.

If both *file1* and *file2* refer to standard input or to the same FIFO special, block special, or character special file, the results are undefined.

STDIN

The standard input shall be used only if one of the *file1* or *file2* operands refers to standard input. See the INPUT FILES section.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *comm*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the collating sequence *comm* expects to have been used when the input files were sorted.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in

arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *comm* utility shall produce output depending on the options selected. If the −**1**, −**2**, and −**3** options are all selected, *comm* shall write nothing to standard output.

If the −**1** option is not selected, lines contained only in *file1* shall be written using the format:

```
"%s\n", <line in file1>
```

If the −**2** option is not selected, lines contained only in *file2* are written using the format:

```
"%s%s\n", <lead>, <line in file2>
```

where the string <*lead*> is as follows:

<tab>	The −**1** option is not selected.
null string	The −**1** option is selected.

If the −**3** option is not selected, lines contained in both files shall be written using the format:

```
"%s%s\n", <lead>, <line in both>
```

where the string <*lead*> is as follows:

<tab><tab>	Neither the −**1** nor the −**2** option is selected.
<tab>	Exactly one of the −**1** and −**2** options is selected.
null string	Both the −**1** and −**2** options are selected.

If the input files were ordered according to the collating sequence of the current locale, the lines written shall be in the collating sequence of the original lines.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were successfully output as specified.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If the input files are not properly presorted, the output of *comm* might not be useful.

EXAMPLES

If a file named **xcu** contains a sorted list of the utilities in this volume of IEEE Std 1003.1-200x, a file named **xpg3** contains a sorted list of the utilities specified in the X/Open Portability Guide, Issue 3, and a file named **svid89** contains a sorted list of the utilities in the System V Interface Definition Third Edition:

```
comm −23 xcu xpg3 | comm −23 − svid89
```

would print a list of utilities in this volume of IEEE Std 1003.1-200x not specified by either of the other documents:

```
comm −12 xcu xpg3 | comm −12 − svid89
```

would print a list of utilities specified by all three documents, and:

```
comm −12 xpg3 svid89 | comm −23 − xcu
```

would print a list of utilities specified by both XPG3 and the SVID, but not specified in this volume of IEEE Std 1003.1-200x.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cmp, *diff*, *sort*, *uniq*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

command — execute a simple command

SYNOPSIS

`command [−p] command_name [argument ...]`

UP `command [ −v | −V ] command_name`

DESCRIPTION

The *command* utility shall cause the shell to treat the arguments as a simple command, suppressing the shell function lookup that is described in Section 2.9.1.1 (on page 2249), item 1b.

If the *command_name* is the same as the name of one of the special built-in utilities, the special properties in the enumerated list at the beginning of Section 2.14 (on page 2266) shall not occur. In every other respect, if *command_name* is not the name of a function, the effect of *command* (with no options) shall be the same as omitting *command*.

On systems supporting the User Portability Utilities option, the *command* utility also shall provide information concerning how a command name is interpreted by the shell; see **−v** and **−V**.

OPTIONS

The *command* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−p Perform the command search using a default value for *PATH* that is guaranteed to find all of the standard utilities.

−v (On systems supporting the User Portability Utilities option.) Write a string to standard output that indicates the pathname or command that will be used by the shell, in the current shell execution environment (see Section 2.12 (on page 2263)), to invoke *command_name*, but do not invoke *command_name*.

- Utilities, regular built-in utilities, *command_name*s including a slash character, and any implementation-defined functions that are found using the *PATH* variable (as described in Section 2.9.1.1 (on page 2249)), shall be written as absolute pathnames.
- Shell functions, special built-in utilities, regular built-in utilities not associated with a *PATH* search, and shell reserved words shall be written as just their names.
- An alias shall be written as a command line that represents its alias definition.
- Otherwise, no output shall be written and the exit status shall reflect that the name was not found.

−V (On systems supporting the User Portability Utilities option.) Write a string to standard output that indicates how the name given in the *command_name* operand will be interpreted by the shell, in the current shell execution environment (see Section 2.12 (on page 2263)), but do not invoke *command_name*. Although the format of this string is unspecified, it shall indicate in which of the following categories *command_name* falls and shall include the information stated:

- Utilities, regular built-in utilities, and any implementation-defined functions that are found using the *PATH* variable (as described in Section 2.9.1.1 (on page 2249)), shall be identified as such and include the absolute pathname in the string.

- Other shell functions shall be identified as functions.
- Aliases shall be identified as aliases and their definitions included in the string.
- Special built-in utilities shall be identified as special built-in utilities.
- Regular built-in utilities not associated with a *PATH* search shall be identified as regular built-in utilities. (The term ''regular'' need not be used.)
- Shell reserved words shall be identified as reserved words.

OPERANDS

The following operands shall be supported:

argument	One of the strings treated as an argument to *command_name*.
command_name	The name of a utility or a special built-in utility.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *command*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the search path used during the command search described in Section 2.9.1.1 (on page 2249), except as described under the −**p** option.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the −**v** option is specified, standard output shall be formatted as:

```
"%s\n", <pathname or command>
```

When the −**V** option is specified, standard output shall be formatted as:

```
"%s\n", <unspecified>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

When the −**v** or −**V** options are specified, the following exit values shall be returned:

0 Successful completion.

>0 The *command_name* could not be found or an error occurred.

Otherwise, the following exit values shall be returned:

126 The utility specified by *command_name* was found but could not be invoked.

127 An error occurred in the *command* utility or the utility specified by *command_name* could not be found.

Otherwise, the exit status of *command* shall be that of the simple command specified by the arguments to *command*.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The order for command search allows functions to override regular built-ins and path searches. This utility is necessary to allow functions that have the same name as a utility to call the utility (instead of a recursive call to the function).

The system default path is available using *getconf*; however, since *getconf* may need to have the *PATH* set up before it can be called itself, the following can be used:

```
command −p getconf _CS_PATH
```

There are some advantages to suppressing the special characteristics of special built-ins on occasion. For example:

```
command exec > unwritable-file
```

does not cause a non-interactive script to abort, so that the output status can be checked by the script.

The *command*, *env*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

Since the −**v** and −**V** options of *command* produce output in relation to the current shell execution environment, *command* is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(PATH=foo command −v)
 nohup command −v
```

it does not necessarily produce correct results. For example, when called with *nohup* or an *exec* function, in a separate utility execution environment, most implementations are not able to identify aliases, functions, or special built-ins.

Two types of regular built-ins could be encountered on a system and these are described separately by *command*. The description of command search in Section 2.9.1.1 (on page 2249) allows for a standard utility to be implemented as a regular built-in as long as it is found in the appropriate place in a *PATH* search. So, for example, *command* −**v** *true* might yield **/bin/true** or some similar pathname. Other implementation-defined utilities that are not defined by this volume of IEEE Std 1003.1-200x might exist only as built-ins and have no pathname associated with them. These produce output identified as (regular) built-ins. Applications encountering these are not able to count on *exec*ing them, using them with *nohup*, overriding them with a different *PATH*, and so on.

EXAMPLES

1. Make a version of *cd* that always prints out the new working directory exactly once:

```
cd() {
    command cd "$@" >/dev/null
    pwd
}
```

2. Start off a ‘‘secure shell script’’ in which the script avoids being spoofed by its parent:

```
IFS='
'
#    The preceding value should be <space><tab><newline>.
#    Set IFS to its default value.

\unalias −a
#    Unset all possible aliases.
#    Note that unalias is escaped to prevent an alias
#    being used for unalias.

unset −f command
#    Ensure command is not a user function.

PATH="$(command −p getconf _CS_PATH):$PATH"
#    Put on a reliable PATH prefix.

#    ...
```

At this point, given correct permissions on the directories called by *PATH*, the script has the ability to ensure that any utility it calls is the intended one. It is being very cautious because it assumes that implementation extensions may be present that would allow user functions to exist when it is invoked; this capability is not specified by this volume of IEEE Std 1003.1-200x, but it is not prohibited as an extension. For example, the *ENV* variable precedes the invocation of the script with a user start-up script. Such a script could define functions to spoof the application.

RATIONALE

Since *command* is a regular built-in utility it is always found prior to the *PATH* search.

There is nothing in the description of *command* that implies the command line is parsed any differently from that of any other simple command. For example:

```
command a | b ; c
```

is not parsed in any special way that causes `'|'` or `';'` to be treated other than a pipe operator or semicolon or that prevents function lookup on **b** or **c**.

The *command* utility is somewhat similar to the Eighth Edition shell *builtin* command, but since *command* also goes to the file system to search for utilities, the name *builtin* would not be intuitive.

The *command* utility is most likely to be provided as a regular built-in. It is not listed as a special built-in for the following reasons:

- The removal of exportable functions made the special precedence of a special built-in unnecessary.
- A special built-in has special properties (see Section 2.14 (on page 2266)) that were inappropriate for invoking other utilities. For example, two commands such as:

  ```
  date > unwritable-file
  ```

  ```
  command date > unwritable-file
  ```

  would have entirely different results; in a non-interactive script, the former would continue to execute the next command, the latter would abort. Introducing this semantic difference along with suppressing functions was seen to be non-intuitive.

The −**p** option is present because it is useful to be able to ensure a safe path search that finds all the POSIX Shell and Utilities standard utilities. This search might not be identical to the one that occurs through one of the POSIX System Interfaces *exec* functions when *PATH* is unset. At the very least, this feature is required to allow the script to access the correct version of *getconf* so that the value of the default path can be accurately retrieved.

The *command* −**v** and −**V** options were added to satisfy requirements from users that are currently accomplished by three different historical utilities: *type* in the System V shell, *whence* in the KornShell, and *which* in the C shell. Since there is no historical agreement on how and what to accomplish here, the POSIX *command* utility was enhanced and the historical utilities were left unmodified. The C shell *which* merely conducts a path search. The KornShell *whence* is more elaborate—in addition to the categories required by POSIX, it also reports on tracked aliases, exported aliases, and undefined functions.

The output format of −**V** was left mostly unspecified because human users are its only audience. Applications should not be written to care about this information; they can use the output of −**v** to differentiate between various types of commands, but the additional information that may be emitted by the more verbose −**V** is not needed and should not be arbitrarily constrained in its verbosity or localization for application parsing reasons.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, *type*

CHANGE HISTORY

First released in Issue 4.

NAME

compress — compress data

SYNOPSIS

XSI

```
compress [−fv][−b bits][file ...]

compress [−cfv][−b bits][file]
```

DESCRIPTION

The *compress* utility shall attempt to reduce the size of the named files by using adaptive Lempel-Ziv coding algorithm.

Note: Lempel-Ziv is US Patent 4464650, issued to William Eastman, Abraham Lempel, Jacob Ziv, Martin Cohn on August 7th, 1984, and assigned to Sperry Corporation. |

Lempel-Ziv-Welch compression is covered by US Patent 4558302, issued to Terry A. Welch on December 10th, 1985, and assigned to Sperry Corporation. |

On systems not supporting adaptive Lempel-Ziv coding algorithm, the input files shall not be changed and an error value greater than two shall be returned. Except when the output is to the standard output, each file shall be replaced by one with the extension **.Z**. If the invoking process has appropriate privileges, the ownership, modes, access time, and modification time of the original file are preserved. If appending the **.Z** to the filename would make the name exceed {NAME_MAX} bytes, the command shall fail. If no files are specified, the standard input shall be compressed to the standard output.

OPTIONS

The *compress* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−b *bits* Specify the maximum number of bits to use in a code. For a conforming application, the *bits* argument shall be: |

`9 <= bits <= 14`

The implementation may allow *bits* values of greater than 14. The default is 14, 15, or 16.

−c Cause *compress* to write to the standard output; the input file is not changed, and no **.Z** files are created.

−f Force compression of *file*, even if it does not actually reduce the size of the file, or if the corresponding *file*.**Z** file already exists. If the −**f** option is not given, and the process is not running in the background, the user is prompted as to whether an existing *file*.**Z** file should be overwritten.

−v Write the percentage reduction of each file to standard error.

OPERANDS

The following operand shall be supported:

file A pathname of a file to be compressed.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'−'`.

INPUT FILES

If *file* operands are specified, the input files contain the data to be compressed.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *compress*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If no *file* operands are specified, or if a *file* operand is **'−'**, or if the −**c** option is specified, the standard output contains the compressed output.

STDERR

The standard error shall be used only for diagnostic and prompt messages and the output from −**v**.

OUTPUT FILES

The output files shall contain the compressed output. The format of compressed files is unspecified and interchange of such files between implementations (including access via unspecified file sharing mechanisms) is not required by IEEE Std 1003.1-200x.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
1	An error occurred.
2	One or more files were not compressed because they would have increased in size (and the −**f** option was not specified).
>2	An error occurred.

CONSEQUENCES OF ERRORS

The input file shall remain unmodified.

APPLICATION USAGE

The amount of compression obtained depends on the size of the input, the number of *bits* per code, and the distribution of common substrings. Typically, text such as source code or English is reduced by 50-60%. Compression is generally much better than that achieved by Huffman coding or adaptive Huffman coding (*compact*), and takes less time to compute.

Although *compress* strictly follows the default actions upon receipt of a signal or when an error occurs, some unexpected results may occur. In some implementations it is likely that a partially compressed file is left in place, alongside its uncompressed input file. Since the general operation of *compress* is to delete the uncompressed file only after the **.Z** file has been successfully filled, an application should always carefully check the exit status of *compress* before arbitrarily deleting files that have like-named neighbors with **.Z** suffixes.

The limit of 14 on the *bits* option-argument is to achieve portability to all systems (within the restrictions imposed by the lack of an explicit published file format). Some implementations based on 16-bit architectures cannot support 15 or 16-bit uncompression. |

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

uncompress, *zcat*

CHANGE HISTORY

First released in Issue 4.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

An error case is added for systems not supporting adaptive Lempel-Ziv coding.

NAME

cp — copy files

SYNOPSIS

```
cp [-fip] source_file target_file

cp [-fip] source_file ... target

cp -R [-H | -L | -P][-fip] source_file ... target

OB  cp -r [-H | -L | -P][-fip] source_file ... target
```

DESCRIPTION

The first synopsis form is denoted by two operands, neither of which are existing files of type directory. The *cp* utility shall copy the contents of *source_file* (or, if *source_file* is a file of type symbolic link, the contents of the file referenced by *source_file*) to the destination path named by *target_file*.

The second synopsis form is denoted by two or more operands where the –**R** or –**r** options are not specified and the first synopsis form is not applicable. It shall be an error if any *source_file* is a file of type directory, if *target* does not exist, or if *target* is a file of a type defined by the System Interfaces volume of IEEE Std 1003.1-200x, but is not a file of type directory. The *cp* utility shall copy the contents of each *source_file* (or, if *source_file* is a file of type symbolic link, the contents of the file referenced by *source_file*) to the destination path named by the concatenation of *target*, a slash character, and the last component of *source_file*.

The third and fourth synopsis forms are denoted by two or more operands where the –**R** or –**r** options are specified. The *cp* utility shall copy each file in the file hierarchy rooted in each *source_file* to a destination path named as follows:

- If *target* exists and is a file of type directory, the name of the corresponding destination path for each file in the file hierarchy shall be the concatenation of *target*, a slash character, and the pathname of the file relative to the directory containing *source_file*.
- If *target* does not exist and two operands are specified, the name of the corresponding destination path for *source_file* shall be *target*; the name of the corresponding destination path for all other files in the file hierarchy shall be the concatenation of *target*, a slash character, and the pathname of the file relative to *source_file*.

It shall be an error if *target* does not exist and more than two operands are specified, or if *target* exists and is a file of a type defined by the System Interfaces volume of IEEE Std 1003.1-200x, but is not a file of type directory.

In the following description, the term *dest_file* refers to the file named by the destination path. The term *source_file* refers to the file that is being copied, whether specified as an operand or a file in a file hierarchy rooted in a *source_file* operand. If *source_file* is a file of type symbolic link:

- If neither the –**R** nor –**r** options were specified, *cp* shall take actions based on the type and contents of the file referenced by the symbolic link, and not by the symbolic link itself.
- If the –**R** option was specified:
  - — If none of the options –**H**, –**L**, nor –**P** were specified, it is unspecified which of –**H**, –**L**, or –**P** will be used as a default.
  - — If the –**H** option was specified, *cp* shall take actions based on the type and contents of the file referenced by any symbolic link specified as a *source_file* operand.
  - — If the –**L** option was specified, *cp* shall take actions based on the type and contents of the file referenced by any symbolic link specified as a *source_file* operand or any symbolic

links encountered during traversal of a file hierarchy.

  - If the −**P** option was specified, *cp* shall copy any symbolic link specified as a *source_file* operand and any symbolic links encountered during traversal of a file hierarchy, and shall not follow any symbolic links.

- If the −**r** option was specified, the behavior is implementation-defined.

For each *source_file*, the following steps shall be taken:

1. If *source_file* references the same file as *dest_file*, *cp* may write a diagnostic message to standard error; it shall do nothing more with *source_file* and shall go on to any remaining files.

2. If *source_file* is of type directory, the following steps shall be taken:

   a. If neither the −**R** or −**r** options were specified, *cp* shall write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files.

   b. If *source_file* was not specified as an operand and *source_file* is dot or dot-dot, *cp* shall do nothing more with *source_file* and go on to any remaining files.

   c. If *dest_file* exists and it is a file type not specified by the System Interfaces volume of IEEE Std 1003.1-200x, the behavior is implementation-defined.

   d. If *dest_file* exists and it is not of type directory, *cp* shall write a diagnostic message to standard error, do nothing more with *source_file* or any files below *source_file* in the file hierarchy, and go on to any remaining files.

   e. If the directory *dest_file* does not exist, it shall be created with file permission bits set to the same value as those of *source_file*, modified by the file creation mask of the user if the −**p** option was not specified, and then bitwise-inclusively OR'ed with S_IRWXU. If *dest_file* cannot be created, *cp* shall write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files. It is unspecified if *cp* attempts to copy files in the file hierarchy rooted in *source_file*.

   f. The files in the directory *source_file* shall be copied to the directory *dest_file*, taking the four steps (1 to 4) listed here with the files as *source_file*s.

   g. If *dest_file* was created, its file permission bits shall be changed (if necessary) to be the same as those of *source_file*, modified by the file creation mask of the user if the −**p** option was not specified.

   h. The *cp* utility shall do nothing more with *source_file* and go on to any remaining files.

3. If *source_file* is of type regular file, the following steps shall be taken:

   a. If *dest_file* exists, the following steps shall be taken:

      i. If the −**i** option is in effect, the *cp* utility shall write a prompt to the standard error and read a line from the standard input. If the response is not affirmative, *cp* shall do nothing more with *source_file* and go on to any remaining files.

      ii. A file descriptor for *dest_file* shall be obtained by performing actions equivalent to the *open*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x called using *dest_file* as the *path* argument, and the bitwise-inclusive OR of O_WRONLY and O_TRUNC as the *oflag* argument.

      iii. If the attempt to obtain a file descriptor fails and the −**f** option is in effect, *cp* shall attempt to remove the file by performing actions equivalent to the *unlink*( ) function defined in the System Interfaces volume of

IEEE Std 1003.1-200x called using *dest_file* as the *path* argument. If this attempt succeeds, *cp* shall continue with step 3b.

b. If *dest_file* does not exist, a file descriptor shall be obtained by performing actions equivalent to the *open*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x called using *dest_file* as the *path* argument, and the bitwise-inclusive OR of O_WRONLY and O_CREAT as the *oflag* argument. The file permission bits of *source_file* shall be the *mode* argument.

c. If the attempt to obtain a file descriptor fails, *cp* shall write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files.

d. The contents of *source_file* shall be written to the file descriptor. Any write errors shall cause *cp* to write a diagnostic message to standard error and continue to step 3e.

e. The file descriptor shall be closed.

f. The *cp* utility shall do nothing more with *source_file*. If a write error occurred in step 3d, it is unspecified if *cp* continues with any remaining files. If no write error occurred in step 3d, *cp* shall go on to any remaining files.

4. Otherwise, the following steps shall be taken:

   a. If the **–r** option was specified, the behavior is implementation-defined.

   b. If the **–R** option was specified, the following steps shall be taken:

      i. The *dest_file* shall be created with the same file type as *source_file*.

      ii. If *source_file* is a file of type FIFO, the file permission bits shall be the same as those of *source_file*, modified by the file creation mask of the user if the **–p** option was not specified. Otherwise, the permissions, owner ID, and group ID of *dest_file* are implementation-defined.

      If this creation fails for any reason, *cp* shall write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files.

      iii. If *source_file* is a file of type symbolic link, the pathname contained in *dest_file* shall be the same as the pathname contained in *source_file*.

      If this fails for any reason, *cp* shall write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files.

If the implementation provides additional or alternate access control mechanisms (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.4, File Access Permissions), their effect on copies of files is implementation-defined.

OPTIONS

The *cp* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

–f If a file descriptor for a destination file cannot be obtained, as described in step 3.a.ii., attempt to unlink the destination file and proceed.

–H Take actions based on the type and contents of the file referenced by any symbolic link specified as a *source_file* operand.

–i Write a prompt to standard error before copying to any existing destination file. If the response from the standard input is affirmative, the copy shall be attempted;

otherwise, it shall not.

–L Take actions based on the type and contents of the file referenced by any symbolic link specified as a *source_file* operand or any symbolic links encountered during traversal of a file hierarchy.

–P Take actions on any symbolic link specified as a *source_file* operand or any symbolic link encountered during traversal of a file hierarchy.

–p Duplicate the following characteristics of each source file in the corresponding destination file:

1. The time of last data modification and time of last access. If this duplication fails for any reason, *cp* shall write a diagnostic message to standard error.
2. The user ID and group ID. If this duplication fails for any reason, it is unspecified whether *cp* writes a diagnostic message to standard error.
3. The file permission bits and the S_ISUID and S_ISGID bits. Other, implementation-defined, bits may be duplicated as well. If this duplication fails for any reason, *cp* shall write a diagnostic message to standard error.

If the user ID or the group ID cannot be duplicated, the file permission bits S_ISUID and S_ISGID shall be cleared. If these bits are present in the source file but are not duplicated in the destination file, it is unspecified whether *cp* writes a diagnostic message to standard error.

The order in which the preceding characteristics are duplicated is unspecified. The *dest_file* shall not be deleted if these characteristics cannot be preserved.

–R Copy file hierarchies.

OB **–r** Copy file hierarchies. The treatment of special files is implementation-defined.

Specifying more than one of the mutually-exclusive options **–H**, **–L**, and **–P** shall not be considered an error. The last option specified shall determine the behavior of the utility.

OPERANDS

The following operands shall be supported:

source_file A pathname of a file to be copied.

target_file A pathname of an existing or nonexistent file, used for the output when a single file is copied.

target A pathname of a directory to contain the copied files.

STDIN

The standard input shall be used to read an input line in response to each prompt specified in the STDERR section. Otherwise, the standard input shall not be used.

INPUT FILES

The input files specified as operands may be of any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cp*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_MESSAGES	Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

A prompt shall be written to standard error under the conditions specified in the DESCRIPTION section. The prompt shall contain the destination pathname, but its format is otherwise unspecified. Otherwise, the standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output files may be of any type.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All files were copied successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If *cp* is prematurely terminated by a signal or error, files or file hierarchies may be only partially copied and files and directories may have incorrect permissions or access and modification times.

APPLICATION USAGE

The difference between −**R** and −**r** is in the treatment by *cp* of file types other than regular and directory. The original −**r** flag, for historic reasons, does not handle special files any differently from regular files, but always reads the file and copies its contents. This has obvious problems in the presence of special file types; for example, character devices, FIFOs, and sockets. The −**R** option is intended to recreate the file hierarchy and the −**r** option supports historical practice. It was anticipated that a future version of this volume of IEEE Std 1003.1-200x would deprecate the −**r** option, and for that reason, there has been no attempt to fix its behavior with respect to FIFOs or other file types where copying the file is clearly wrong. However, some implementations support −**r** with the same abilities as the −**R** defined in this volume of IEEE Std 1003.1-200x. To accommodate them as well as systems that do not, the differences between −**r** and −**R** are implementation-defined. Implementations may make them identical. The −**r** option is now marked obsolescent.

The set-user-ID and set-group-ID bits are explicitly cleared when files are created. This is to prevent users from creating programs that are set-user-ID or set-group-ID to them when copying files or to make set-user-ID or set-group-ID files accessible to new groups of users. For example, if a file is set-user-ID and the copy has a different group ID than the source, a new group of users has execute permission to a set-user-ID program than did previously. In particular, this is a problem for superusers copying users' trees.

EXAMPLES

None.

RATIONALE

The −**i** option exists on BSD systems, giving applications and users a way to avoid accidentally removing files when copying. Although the 4.3 BSD version does not prompt if the standard input is not a terminal, the standard developers decided that use of −**i** is a request for interaction, so when the destination path exists, the utility takes instructions from whatever responds on standard input.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application using the −**i** option relies on the system to provide the most suitable dialog directly with the user, based on the behavior specified.

The −**p** option is historical practice on BSD systems, duplicating the time of last data modification and time of last access. This volume of IEEE Std 1003.1-200x extends it to preserve the user and group IDs, as well as the file permissions. This requirement has obvious problems in that the directories are almost certainly modified after being copied. This volume of IEEE Std 1003.1-200x requires that the modification times be preserved. The statement that the order in which the characteristics are duplicated is unspecified is to permit implementations to provide the maximum amount of security for the user. Implementations should take into account the obvious security issues involved in setting the owner, group, and mode in the wrong order or creating files with an owner, group, or mode different from the final value.

It is unspecified whether *cp* writes diagnostic messages when the user and group IDs cannot be set due to the widespread practice of users using −**p** to duplicate some portion of the file characteristics, indifferent to the duplication of others. Historic implementations only write diagnostic messages on errors other than [EPERM].

The −**r** option is historical practice on BSD and BSD-derived systems, copying file hierarchies as opposed to single files. This functionality is used heavily in historical applications, and its loss would significantly decrease consensus. The −**R** option was added as a close synonym to the −**r** option, selected for consistency with all other options in this volume of IEEE Std 1003.1-200x that do recursive directory descent.

When a failure occurs during the copying of a file hierarchy, *cp* is required to attempt to copy files that are on the same level in the hierarchy or above the file where the failure occurred. It is unspecified if *cp* shall attempt to copy files below the file where the failure occurred (which cannot succeed in any case).

Permissions, owners, and groups of created special file types have been deliberately left as implementation-defined. This is to allow systems to satisfy special requirements (for example, allowing users to create character special devices, but requiring them to be owned by a certain group). In general, it is strongly suggested that the permissions, owner, and group be the same as if the user had run the historical *mknod*, *ln*, or other utility to create the file. It is also probable that additional privileges are required to create block, character, or other implementation-defined special file types.

Additionally, the −**p** option explicitly requires that all set-user-ID and set-group-ID permissions be discarded if any of the owner or group IDs cannot be set. This is to keep users from unintentionally giving away special privilege when copying programs.

When creating regular files, historical versions of *cp* use the mode of the source file as modified by the file mode creation mask. Other choices would have been to use the mode of the source file unmodified by the creation mask or to use the same mode as would be given to a new file created by the user (plus the execution bits of the source file) and then modify it by the file mode creation mask. In the absence of any strong reason to change historic practice, it was in large part retained.

When creating directories, historical versions of *cp* use the mode of the source directory, plus read, write, and search bits for the owner, as modified by the file mode creation mask. This is done so that *cp* can copy trees where the user has read permission, but the owner does not. A side effect is that if the file creation mask denies the owner permissions, *cp* fails. Also, once the copy is done, historical versions of *cp* set the permissions on the created directory to be the same as the source directory, unmodified by the file creation mask.

This behavior has been modified so that *cp* is always able to create the contents of the directory, regardless of the file creation mask. After the copy is done, the permissions are set to be the same as the source directory, as modified by the file creation mask. This latter change from historical behavior is to prevent users from accidentally creating directories with permissions beyond those they would normally set and for consistency with the behavior of *cp* in creating files.

It is not a requirement that *cp* detect attempts to copy a file to itself; however, implementations are strongly encouraged to do so. Historical implementations have detected the attempt in most cases.

There are two methods of copying subtrees in this volume of IEEE Std 1003.1-200x. The other method is described as part of the *pax* utility (see *pax* (on page 2900)). Both methods are historical practice. The *cp* utility provides a simpler, more intuitive interface, while *pax* offers a finer granularity of control. Each provides additional functionality to the other; in particular, *pax* maintains the hard-link structure of the hierarchy, while *cp* does not. It is the intention of the standard developers that the results be similar (using appropriate option combinations in both utilities). The results are not required to be identical; there seemed insufficient gain to applications to balance the difficulty of implementations having to guarantee that the results would be exactly identical.

The wording allowing *cp* to copy a directory to implementation-defined file types not specified by the System Interfaces volume of IEEE Std 1003.1-200x is provided so that implementations supporting symbolic links are not required to prohibit copying directories to symbolic links. Other extensions to the System Interfaces volume of IEEE Std 1003.1-200x file types may need to use this loophole as well.

FUTURE DIRECTIONS

The –**r** option may be removed; use –**R** instead.

SEE ALSO

mv, *find*, *ln*, *pax*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The –**r** option is marked obsolescent.

The new options –**H**, –**L**, and –**P** are added to align with the IEEE P1003.2b draft standard. These options affect the processing of symbolic links.

IEEE PASC Interpretation 1003.2 #194 is applied, adding a description of the –**P** option.

NAME

crontab — schedule periodic background work

SYNOPSIS

UP
```
crontab [file]

crontab [ −e | −l | −r ]
```

DESCRIPTION

The *crontab* utility shall create, replace, or edit a user's *crontab* entry; a crontab entry is a list of commands and the times at which they shall be executed. The new crontab entry can be input by specifying *file* or input from standard input if no *file* operand is specified, or by using an editor, if **−e** is specified.

Upon execution of a command from a crontab entry, the implementation shall supply a default environment, defining at least the following environment variables:

HOME	A pathname of the user's home directory.
LOGNAME	The user's login name.
PATH	A string representing a search path guaranteed to find all of the standard utilities.
SHELL	A pathname of the command interpreter. When *crontab* is invoked as specified by this volume of IEEE Std 1003.1-200x, the value shall be a pathname for *sh*.

The values of these variables when *crontab* is invoked as specified by this volume of IEEE Std 1003.1-200x shall not affect the default values provided when the scheduled command is run.

If standard output and standard error are not redirected by commands executed from the crontab entry, any generated output or errors shall be mailed, via an implementation-defined method, to the user.

XSI Users shall be permitted to use *crontab* if their names appear in the file **/usr/lib/cron/cron.allow**. If that file does not exist, the file **/usr/lib/cron/cron.deny** shall be checked to determine whether the user shall be denied access to *crontab*. If neither file exists, only a process with appropriate privileges shall be allowed to submit a job. If only **cron.deny** exists and is empty, global usage shall be permitted. The **cron.allow** and **cron.deny** files shall consist of one user name per line.

OPTIONS

The *crontab* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−e	Edit a copy of the invoking user's crontab entry, or create an empty entry to edit if the crontab entry does not exist. When editing is complete, the entry shall be installed as the user's crontab entry.
−l	(The letter ell.) List the invoking user's crontab entry.
−r	Remove the invoking user's crontab entry.

OPERANDS

The following operand shall be supported:

file	The pathname of a file that contains specifications, in the format defined in the INPUT FILES section, for crontab entries.

STDIN

See the INPUT FILES section.

INPUT FILES

In the POSIX locale, the user or application shall ensure that a crontab entry is a text file consisting of lines of six fields each. The fields shall be separated by <blank>s. The first five fields shall be integer patterns that specify the following:

1. Minute [0,59]
2. Hour [0,23]
3. Day of the month [1,31]
4. Month of the year [1,12]
5. Day of the week ([0,6] with 0=Sunday)

Each of these patterns can be either an asterisk (meaning all valid values), an element, or a list of elements separated by commas. An element shall be either a number or two numbers separated by a hyphen (meaning an inclusive range). The specification of days can be made by two fields (day of the month and day of the week). If month, day of month, and day of week are all asterisks, every day shall be matched. If either the month or day of month is specified as an element or list, but the day of week is an asterisk, the month and day of month fields shall specify the days that match. If both month and day of month are specified as asterisk, but day of week is an element or list, then only the specified days of the week match. Finally, if either the month or day of month is specified as an element or list, and the day of week is also specified as an element or list, then any day matching either the month and day of month, or the day of week, shall be matched.

The sixth field of a line in a crontab entry is a string that shall be executed by *sh* at the specified times. A percent sign character in this field shall be translated to a <newline>. Any character preceded by a backslash (including the '%') shall cause that character to be treated literally. Only the first line (up to a '%' or end-of-line) of the command field shall be executed by the command interpreter. The other lines shall be made available to the command as standard input.

Blank lines and those whose first non-<blank> is '#' shall be ignored.

XSI The text files **/usr/lib/cron/cron.allow** and **/usr/lib/cron/cron.deny** shall contain zero or more user names, one per line, of users who are, respectively, authorized or denied access to the service underlying the *crontab* utility.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *crontab*:

EDITOR Determine the editor to be invoked when the **−e** option is specified. The default editor shall be *vi*.

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the **−l** option is specified, the crontab entry shall be written to the standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

The user's crontab entry is not submitted, removed, edited, or listed.

APPLICATION USAGE

The format of the *crontab* entry shown here is guaranteed only for the POSIX locale. Other cultures may be supported with substantially different interfaces, although implementations are encouraged to provide comparable levels of functionality.

The default settings of the *HOME*, *LOGNAME*, *PATH*, and *SHELL* variables that are given to the scheduled job are not affected by the settings of those variables when *crontab* is run; as stated, they are defaults. The text about ''invoked as specified by this volume of IEEE Std 1003.1-200x'' means that the implementation may provide extensions that allow these variables to be affected at runtime, but that the user has to take explicit action in order to access the extension, such as give a new option flag or modify the format of the crontab entry.

A typical user error is to type only *crontab*; this causes the system to wait for the new crontab entry on standard input. If end-of-file is typed (generally <control>-D), the crontab entry is replaced by an empty file. In this case, the user should type the interrupt character, which prevents the crontab entry from being replaced.

EXAMPLES

1. Clean up **core** files every weekday morning at 3:15 am:

```
15 3 * * 1-5 find $HOME −name core 2>/dev/null | xargs rm −f
```

2. Mail a birthday greeting:

```
0 12 14 2 * mailx john%Happy Birthday!%Time for lunch.
```

3. As an example of specifying the two types of days:

```
0 0 1,15 * 1
```

would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to `'*'`; for example:

```
0 0 * * 1
```

would run a command only on Mondays.

RATIONALE

All references to a *cron* daemon and to *cron files* have been omitted. Although historical implementations have used this arrangement, there is no reason to limit future implementations.

This description of *crontab* is designed to support only users with normal privileges. The format of the input is based on the System V *crontab*; however, there is no requirement here that the actual system database used by the *cron* daemon (or a similar mechanism) use this format internally. For example, systems derived from BSD are likely to have an additional field appended that indicates the user identity to be used when the job is submitted.

The −**e** option was adopted from the SVID as a user convenience, although it does not exist in all historical implementations.

FUTURE DIRECTIONS

None.

SEE ALSO

at

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

csplit — split files based on context

SYNOPSIS

UP
```
csplit [−ks][−f prefix][−n number] file arg1 ...argn
```

DESCRIPTION

The *csplit* utility shall read the file named by the *file* operand, write all or part of that file into other files as directed by the *arg* operands, and write the sizes of the files.

OPTIONS

The *csplit* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−f *prefix*	Name the created files *prefix***00**, *prefix***01**, . . ., *prefixn*. The default is **xx00** . . . **xx***n*. If the *prefix* argument would create a filename exceeding {NAME_MAX} bytes, an error shall result, *csplit* shall exit with a diagnostic message and no files shall be created.
−k	Leave previously created files intact. By default, *csplit* shall remove created files if an error occurs.
−n *number*	Use *number* decimal digits to form filenames for the file pieces. The default shall be 2.
−s	Suppress the output of file size messages.

OPERANDS

The following operands shall be supported:

file	The pathname of a text file to be split. If *file* is '−', the standard input shall be used.

The operands *arg1* . . . *argn* can be a combination of the following:

/*rexp*/**[***offset***]**	A file shall be created using the content of the lines from the current line up to, but not including, the line that results from the evaluation of the regular expression with *offset*, if any, applied. The regular expression *rexp* shall follow the rules for basic regular expressions described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions. The application shall use the sequence "\/" to specify a slash character within the *rexp*. The optional offset shall be a positive or negative integer value representing a number of lines. A positive integer value can be preceded by '+'. If the selection of lines from an *offset* expression of this type would create a file with zero lines, or one with greater than the number of lines left in the input file, the results are unspecified. After the section is created, the current line shall be set to the line that results from the evaluation of the regular expression with any offset applied. If the current line is the first line in the file and a regular expression operation has not yet been performed, the pattern match of *rexp* shall be applied from the current line to the end of the file. Otherwise, the pattern match of *rexp* shall be applied from the line following the current line to the end of the file.
%*rexp*%**[***offset***]**	Equivalent to /*rexp*/**[***offset***]**, except that no file shall be created for the selected section of the input file. The application shall use the sequence "\%" to specify a

percent-sign character within the *rexp*.

line_no	Create a file from the current line up to (but not including) the line number *line_no*. Lines in the file shall be numbered starting at one. The current line becomes *line_no*.
{*num*}	Repeat operand. This operand can follow any of the operands described previously. If it follows a *rexp* type operand, that operand shall be applied *num* more times. If it follows a *line_no* operand, the file shall be split every *line_no* lines, *num* times, from that point.

An error shall be reported if an operand does not reference a line between the current position and the end of the file.

STDIN

See the INPUT FILES section.

INPUT FILES

The input file shall be a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *csplit*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes within regular expressions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

If the −**k** option is specified, created files shall be retained. Otherwise, the default action occurs.

STDOUT

Unless the −**s** option is used, the standard output shall consist of one line per file created, with a format as follows:

```
"%d\n", <file size in bytes>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output files shall contain portions of the original input file; otherwise, unchanged.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

By default, created files shall be removed if an error occurs. When the −**k** option is specified, created files shall not be removed if an error occurs.

APPLICATION USAGE

None.

EXAMPLES

1. This example creates four files, **cobol00** . . . **cobol03**:

   ```
   csplit −f cobol file ’/procedure division/’ /par5./ /par16./
   ```

   After editing the split files, they can be recombined as follows:

   ```
   cat cobol0[0−3] > file
   ```

   Note that this example overwrites the original file.

2. This example would split the file after the first 99 lines, and every 100 lines thereafter, up to 9 999 lines; this is because lines in the file are numbered from 1 rather than zero, for historical reasons:

   ```
   csplit −k file  100  {99}
   ```

3. Assuming that **prog.c** follows the C-language coding convention of ending routines with a `’}’` at the beginning of the line, this example creates a file containing each separate C routine (up to 21) in **prog.c**:

   ```
   csplit −k prog.c ’%main(%’  ’/ˆ}/+1’ {20}
   ```

RATIONALE

The −**n** option was added to extend the range of filenames that could be handled.

Consideration was given to adding a −**a** flag to use the alphabetic filename generation used by the historical *split* utility, but the functionality added by the −**n** option was deemed to make alphabetic naming unnecessary.

FUTURE DIRECTIONS

None.

SEE ALSO

sed, *split*

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

The description of regular expression operands is changed to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

ctags — create a tags file (**DEVELOPMENT**, **FORTRAN**)

SYNOPSIS

UP
```
ctags [−a][−f tagsfile] pathname ...

ctags −x pathname ...
```

DESCRIPTION

The *ctags* utility shall be provided on systems that support the User Portability Utilities option, the Software Development Utilities option, and either or both of the C-Language Development Utilities option and FORTRAN Development Utilities option. On other systems, it is optional.

The *ctags* utility shall write a *tags* file or an index of objects from C-language or FORTRAN source files specified by the *pathname* operands. The tags file shall list the locators of language-specific objects within the source files. A locator consists of a name, pathname, and either a search pattern or a line number that can be used in searching for the object definition. The objects that shall be recognized are specified in the EXTENDED DESCRIPTION section.

OPTIONS

The *ctags* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Append to tags file.
−f *tagsfile*	Write the object locator lists into *tagsfile* instead of the default file named **tags** in the current directory.
−x	Produce a list of object names, the line number, and filename in which each is defined, as well as the text of that line, and write this to the standard output. A **tags** file shall not be created when **−x** is specified.

OPERANDS

The following *pathname* operands are supported:

file.**c**	Files with basenames ending with the **.c** suffix shall be treated as C-language source code. Such files that are not valid input to *c99* produce unspecified results.
file.**h**	Files with basenames ending with the **.h** suffix shall be treated as C-language source code. Such files that are not valid input to *c99* produce unspecified results.
file.**f**	Files with basenames ending with the **.f** suffix shall be treated as FORTRAN-language source code. Such files that are not valid input to *fort77* produce unspecified results.

The handling of other files is implementation-defined.

STDIN

See the INPUT FILES section.

INPUT FILES

The input files shall be text files containing source code in the language indicated by the operand filename suffixes.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ctags*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the order in which output is sorted for the −**x** option. The POSIX locale determines the order in which the tags file is written.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files). When processing C-language source code, if the locale is not compatible with the C locale described by the ISO C standard, the results are unspecified.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The list of object name information produced by the −**x** option shall be written to standard output in the following format:

```
"%s %d %s %s", <object-name>, <line-number>, <filename>,
    <text>
```

where <*text*> is the text of line <*line-number*> of file <*filename*>.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

When the −**x** option is not specified, the format of the output file shall be:

```
"%s\t%s\t/%s/\n", <identifier>, <filename>, <pattern>
```

where <*pattern*> is a search pattern that could be used by an editor to find the defining instance of <*identifier*> in <*filename*> (where *defining instance* is indicated by the declarations listed in the EXTENDED DESCRIPTION).

An optional circumflex ('^') can be added as a prefix to <*pattern*>, and an optional dollar sign can be appended to <*pattern*> to indicate that the pattern is anchored to the beginning (end) of a line of text. Any slash or backslash characters in <*pattern*> shall be preceded by a backslash character. The anchoring circumflex, dollar sign, and escaping backslash characters shall not be considered part of the search pattern. All other characters in the search pattern shall be considered literal characters.

An alternative format is:

```
"%s\t%s\t?%s?\n", <identifier>, <filename>, <pattern>
```

which is identical to the first format except that slashes in <*pattern*> shall not be preceded by escaping backslash characters, and question mark characters in <*pattern*> shall be preceded by backslash characters.

A second alternative format is:

```
"%s\t%s\t%d\n", <identifier>, <filename>, <lineno>
```

where <*lineno*> is a decimal line number that could be used by an editor to find <*identifier*> in <*filename*>.

Neither alternative format shall be produced by *ctags* when it is used as described by IEEE Std 1003.1-200x, but the standard utilities that process tags files shall be able to process those formats as well as the first format.

In any of these formats, the file shall be sorted by identifier, based on the collation sequence in the POSIX locale.

EXTENDED DESCRIPTION

If the operand identifies C-language source, the *ctags* utility shall attempt to produce an output line for each of the following objects:

- Function definitions
- Type definitions
- Macros with arguments

It may also produce output for any of the following objects:

- Function prototypes
- Structures
- Unions
- Global variable definitions
- Enumeration types
- Macros without arguments
- **#define** statements
- **#line** statements

Any **#if** and **#ifdef** statements shall produce no output. The tag **main** is treated specially in C programs. The tag formed shall be created by prefixing **M** to the name of the file, with the trailing **.c**, and leading pathname components (if any) removed.

On systems that do not support the C-Language Development Utilities option, *ctags* produces undefined results for C-language source code files. It should write to standard error a message identifying this condition and cause a non-zero exit status to be produced.

If the operand identifies FORTRAN source, the *ctags* utility shall produce an output line for each function definition. It may also produce output for any of the following objects:

- Subroutine definitions
- COMMON statements
- PARAMETER statements

- DATA and BLOCK DATA statements
- Statement numbers

On systems that do not support the FORTRAN Development Utilities option, *ctags* produces unspecified results for FORTRAN source code files. It should write to standard error a message identifying this condition and cause a non-zero exit status to be produced.

It is implementation-defined what other objects (including duplicate identifiers) produce output.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The output with **–x** is meant to be a simple index that can be written out as an off-line readable function index. If the input files to *ctags* (such as **.c** files) were not created using the same locale as that in effect when *ctags* **–x** is run, results might not be as expected.

The description of C-language processing says ‘‘attempts to’’ because the C language can be greatly confused, especially through the use of **#define**s, and this utility would be of no use if the real C preprocessor were run to identify them. The output from *ctags* may be fooled and incorrect for various constructs.

EXAMPLES

None.

RATIONALE

The option list was significantly reduced from that provided by historical implementations. The **–F** option was omitted as redundant, since it is the default. The **–B** option was omitted as being of very limited usefulness. The **–t** option was omitted since the recognition of typedefs is now required for C source files. The **–u** option was omitted because the update function was judged to be not only inefficient, but also rarely needed.

An early proposal included a **–w** option to suppress warning diagnostics. Since the types of such diagnostics could not be described, the option was omitted as being not useful.

The text for *LC_CTYPE* about compatibility with the C locale acknowledges that the ISO C standard imposes requirements on the locale used to process C source. This could easily be a superset of that known as ‘‘the C locale’’ by way of implementation extensions, or one of a few alternative locales for systems supporting different codesets. No statement is made for FORTRAN because the ANSI X3.9-1978 standard (FORTRAN 77) does not (yet) define a similar locale concept. However, a general rule in this volume of IEEE Std 1003.1-200x is that any time that locales do not match (preparing a file for one locale and processing it in another), the results are suspect.

The collation sequence of the tags file is not affected by *LC_COLLATE* because it is typically not used by human readers, but only by programs such as *vi* to locate the tag within the source files. Using the POSIX locale eliminates some of the problems of coordinating locales between the *ctags* file creator and the *vi* file reader.

Historically, the tags file has been used only by *ex* and *vi*. However, the format of the tags file has been published to encourage other programs to use the tags in new ways. The format allows either patrerns or line numbers to find the identifiers because the historical *vi* recognizes either.

The *ctags* utility does not produce the format using line numbers because it is not useful following any source file changes that add or delete lines. The documented search patterns match historical practice. It should be noted that literal leading circumflex or trailing dollar-sign characters in the search pattern will only behave correctly if anchored to the beginning of the line or end of the line by an additional circumflex or dollar-sign character.

Historical implementations also understand the objects used by the languages Pascal and sometimes LISP, and they understand the C source output by *lex* and *yacc*. The *ctags* utility is not required to accommodate these languages, although implementors are encouraged to do so.

The following historical option was not specified, as *vgrind* is not included in this volume of IEEE Std 1003.1-200x:

–**v** If the –**v** flag is given, an index of the form expected by *vgrind* is produced on the standard output. This listing contains the function name, filename, and page number (assuming 64-line pages). Since the output is sorted into lexicographic order, it may be desired to run the output through *sort* –**f**. Sample use:

```
ctags –v files | sort –f > index vgrind –x index
```

The special treatment of the tag **main** makes the use of *ctags* practical in directories with more than one program.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *fort77*, *vi*

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The OUTPUT FILES section is changed to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

IEEE PASC Interpretation 1003.2 #168 is applied, changing ‘‘create’’ to ‘‘write’’ in the DESCRIPTION.

NAME

cut — cut out selected fields of each line of a file

SYNOPSIS

```
cut –b list [–n] [file ...]

cut –c list [file ...]

cut –f list [–d delim][–s][file ...]
```

DESCRIPTION

The *cut* utility shall cut out bytes (–**b** option), characters (–**c** option) or character-delimited fields (–**f** option) from each line in one or more files, concatenate them, and write them to standard output.

OPTIONS

The *cut* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The application shall ensure that the option-argument *list* (see options –**b**, –**c**, and –**f** below) is a comma-separated list or <blank>-separated list of positive numbers and ranges. Ranges can be in three forms. The first is two positive numbers separated by a hyphen (*low*–*high*), which represents all fields from the first number to the second number. The second is a positive number preceded by a hyphen (–*high*), which represents all fields from field number 1 to that number. The third is a positive number followed by a hyphen (*low*–), which represents that number to the last field, inclusive. The elements in *list* can be repeated, can overlap, and can be specified in any order, but the bytes, characters, or fields selected shall be written in the order of the input data. If an element appears in the selection list more than once, it shall be written exactly once.

The following options shall be supported:

–**b** *list* Cut based on a *list* of bytes. Each selected byte shall be output unless the –**n** option is also specified. It shall not be an error to select bytes not present in the input line.

–**c** *list* Cut based on a *list* of characters. Each selected character shall be output. It shall not be an error to select characters not present in the input line.

–**d** *delim* Set the field delimiter to the character *delim*. The default is the <tab>.

–**f** *list* Cut based on a *list* of fields, assumed to be separated in the file by a delimiter character (see –**d**). Each selected field shall be output. Output fields shall be separated by a single occurrence of the field delimiter character. Lines with no field delimiters shall be passed through intact, unless –**s** is specified. It shall not be an error to select fields not present in the input line.

–**n** Do not split characters. When specified with the –**b** option, each element in *list* of the form *low*–*high* (hyphen-separated numbers) shall be modified as follows:

- If the byte selected by *low* is not the first byte of a character, *low* shall be decremented to select the first byte of the character originally selected by *low*. If the byte selected by *high* is not the last byte of a character, *high* shall be decremented to select the last byte of the character prior to the character originally selected by *high*, or zero if there is no prior character. If the resulting range element has *high* equal to zero or *low* greater than *high*, the list element shall be dropped from *list* for that input line without causing an error.

Each element in *list* of the form *low*– shall be treated as above with *high* set to the number of bytes in the current line, not including the terminating <newline>. Each element in *list* of the form −*high* shall be treated as above with *low* set to 1. Each element in *list* of the form *num* (a single number) shall be treated as above with *low* set to *num* and *high* set to *num*.

−s	Suppress lines with no delimiter characters, when used with the **−f** option. Unless specified, lines with no delimiters shall be passed through untouched.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If no *file* operands are specified, or if a *file* operand is `'−'`, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'−'`. See the INPUT FILES section.

INPUT FILES

The input files shall be text files, except that line lengths shall be unlimited.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cut*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *cut* utility output shall be a concatenation of the selected bytes, characters, or fields (one of the following):

```
"%s\n", <concatenation of bytes>

"%s\n", <concatenation of characters>

"%s\n", <concatenation of fields and field delimiters>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were output successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Earlier versions of the *cut* utility worked in an environment where bytes and characters were considered equivalent (modulo <backspace> and <tab> processing in some implementations). In the extended world of multi-byte characters, the new –**b** option has been added. The –**n** option (used with –**b**) allows it to be used to act on bytes rounded to character boundaries. The algorithm specified for –**n** guarantees that:

```
cut –b 1–500 –n file > file1
cut –b 501– –n file > file2
```

ends up with all the characters in **file** appearing exactly once in **file1** or **file2**. (There is, however, a <newline> in both **file1** and **file2** for each <newline> in **file**.)

EXAMPLES

Examples of the option qualifier list:

1,4,7	Select the first, fourth, and seventh bytes, characters, or fields and field delimiters.
1–3,8	Equivalent to 1,2,3,8.
–5,10	Equivalent to 1,2,3,4,5,10.
3–	Equivalent to third to last, inclusive.

The *low*–*high* forms are not always equivalent when used with –**b** and –**n** and multi-byte characters; see the description of –**n**.

The following command:

```
cut –d : –f 1,6 /etc/passwd
```

reads the System V password file (user database) and produces lines of the form:

```
<user ID>:<home directory>
```

Most utilities in this volume of IEEE Std 1003.1-200x work on text files. The *cut* utility can be used to turn files with arbitrary line lengths into a set of text files containing the same data. The *paste* utility can be used to create (or recreate) files with arbitrary line lengths. For example, if **file** contains long lines:

```
cut –b 1–500 –n file > file1
cut –b 501– –n file > file2
```

creates **file1** (a text file) with lines no longer than 500 bytes (plus the <newline>) and **file2** that contains the remainder of the data from **file**. (Note that **file2** is not a text file if there are lines in **file** that are longer than 500 + {LINE_MAX} bytes.) The original file can be recreated from **file1** and **file2** using the command:

```
paste –d "\0" file1 file2 > file
```

RATIONALE

Some historical implementations do not count <backspace>s in determining character counts with the **−c** option. This may be useful for using *cut* for processing *nroff* output. It was deliberately decided not to have the **−c** option treat either <backspace>s or <tab>s in any special fashion. The *fold* utility does treat these characters specially.

Unlike other utilities, some historical implementations of *cut* exit after not finding an input file, rather than continuing to process the remaining *file* operands. This behavior is prohibited by this volume of IEEE Std 1003.1-200x, where only the exit status is affected by this problem.

The behavior of *cut* when provided with either mutually-exclusive options or options that do not work logically together has been deliberately left unspecified in favor of global wording in Section 1.11 (on page 2221).

The OPTIONS section was changed in response to P1003.2-N149. The change represents historical practice on all known systems. The original standard was ambiguous on the nature of the output.

The *list* option-arguments are historically used to select the portions of the line to be written, but do not affect the order of the data. For example:

```
echo abcdefghi | cut −c6,2,4−7,1
```

yields `"abdefg"`.

A proposal to enhance *cut* with the following option:

−o Preserve the selected field order. When this option is specified, each byte, character, or field (or ranges of such) shall be written in the order specified by the *list* option-argument, even if this requires multiple outputs of the same bytes, characters, or fields.

was rejected because this type of enhancement is outside the scope of the IEEE P1003.2b draft standard.

FUTURE DIRECTIONS

None.

SEE ALSO

grep, *paste*, Section 2.5 (on page 2235)

CHANGE HISTORY

First released in Issue 2.

Issue 6

The OPTIONS section is changed to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

cxref — generate a C-language program cross-reference table (**DEVELOPMENT**)

SYNOPSIS

XSI
```
cxref [−cs][−o file][−w num] [−D name[=def]]...[−I dir]...
    [−U name]... file ...
```

DESCRIPTION

The *cxref* utility shall analyze a collection of C-language *file*s and attempt to build a cross-reference table. Information from **#define** lines shall be included in the symbol table. A sorted listing shall be written to standard output of all symbols (auto, static, and global) in each *file* separately, or with the −**c** option, in combination. Each symbol shall contain an asterisk before the declaring reference.

OPTIONS

The *cxref* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the order of the −**D**, −**I**, and −**U** options (which are identical to their interpretation by *c99*) is significant. The following options shall be supported:

−**c**	Write a combined cross-reference of all input files.
−**s**	Operate silently; do not print input filenames.
−**o** *file*	Direct output to named *file*.
−**w** *num*	Format output no wider than *num* (decimal) columns. This option defaults to 80 if *num* is not specified or is less than 51.
−**D**	Equivalent to *c99*.
−**I**	Equivalent to *c99*.
−**U**	Equivalent to *c99*.

OPERANDS

The following operand shall be supported:

file	A pathname of a C-language source file.

STDIN

Not used.

INPUT FILES

The input files are C-language source files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *cxref*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the ordering of the output.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in

arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be used for the cross-reference listing, unless the −**o** option is used to select a different output file.

The format of standard output is unspecified, except that the following information shall be included:

- If the −**c** option is not specified, each portion of the listing shall start with the name of the input file on a separate line.
- The name line shall be followed by a sorted list of symbols, each with its associated location pathname, the name of the function in which it appears (if it is not a function name itself), and line number references.
- Each line number may be preceded by an asterisk (`'*'`) flag, meaning that this is the declaring reference. Other single-character flags, with implementation-defined meanings, may be included.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output file named by the −**o** option shall be used instead of standard output.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

c99

CHANGE HISTORY

First released in Issue 2.

Issue 5

In the SYNOPSIS, [−**U** *dir*] is changed to [−**U** *name*].

Issue 6

The APPLICATION USAGE section is added.

NAME

date — write the date and time

SYNOPSIS

```
date [−u] [+format]
```

XSI
```
date [−u] mmddhhmm[[cc]yy]
```

DESCRIPTION

XSI The *date* utility shall write the date and time to standard output or attempt to set the system date and time. By default, the current date and time shall be written. If an operand beginning with `'+'` is specified, the output format of *date* shall be controlled by the conversion specifications and other text in the operand.

OPTIONS

The *date* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−u Perform operations as if the *TZ* environment variable was set to the string `"UTC0"`, or its equivalent historical value of `"GMT0"`. Otherwise, *date* shall use the timezone indicated by the *TZ* environment variable or the system default if that variable is unset or null.

OPERANDS

The following operands shall be supported:

+format When the format is specified, each conversion specifier shall be replaced in the standard output by its corresponding value. All other characters shall be copied to the output without change. The output always shall be terminated with a <newline>.

Conversion Specifications

`%a`	Locale's abbreviated weekday name.
`%A`	Locale's full weekday name.
`%b`	Locale's abbreviated month name.
`%B`	Locale's full month name.
`%c`	Locale's appropriate date and time representation.
`%C`	Century (a year divided by 100 and truncated to an integer) as a decimal number [00,99].
`%d`	Day of the month as a decimal number [01,31].
`%D`	Date in the format *mm*/*dd*/*yy*.
`%e`	Day of the month as a decimal number [1,31] in a two-digit field with leading space character fill.
`%h`	A synonym for `%b`.
`%H`	Hour (24-hour clock) as a decimal number [00,23].
`%I`	Hour (12-hour clock) as a decimal number [01,12].

`%j`	Day of the year as a decimal number [001,366].
`%m`	Month as a decimal number [01,12].
`%M`	Minute as a decimal number [00,59].
`%n`	A <newline>.
`%p`	Locale's equivalent of either AM or PM.
`%r`	12-hour clock time [01,12] using the AM/PM notation; in the POSIX locale, this shall be equivalent to `%I:%M:%S %p`.
`%S`	Seconds as a decimal number [00,60].
`%t`	A <tab>.
`%T`	24-hour clock time [00,23] in the format *HH*:*MM*:*SS*.
`%u`	Weekday as a decimal number [1,7] (1=Monday).
`%U`	Week of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday shall be considered to be in week 0.
`%V`	Week of the year (Monday as the first day of the week) as a decimal number [01,53]. If the week containing January 1 has four or more days in the new year, then it shall be considered week 1; otherwise, it shall be the last week of the previous year, and the next week shall be week 1.
`%w`	Weekday as a decimal number [0,6] (0=Sunday).
`%W`	Week of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday shall be considered to be in week 0.
`%x`	Locale's appropriate date representation.
`%X`	Locale's appropriate time representation.
`%y`	Year within century [00,99].
`%Y`	Year with century as a decimal number.
`%Z`	Timezone name, or no characters if no timezone is determinable.
`%%`	A percent sign character.

See the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5, LC_TIME for the conversion specifier values in the POSIX locale.

Modified Conversion Specifications

Some conversion specifiers can be modified by the `E` and `O` modifier characters to indicate a different format or specification as specified in the *LC_TIME* locale description (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5, LC_TIME). If the corresponding keyword (see **era**, **era_year**, **era_d_fmt**, and **alt_digits** in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3.5, LC_TIME) is not specified or not supported for the current locale, the unmodified conversion specifier value shall be used.

`%Ec`	Locale's alternative appropriate date and time representation.

`%EC`	The name of the base year (period) in the locale's alternative representation.
`%Ex`	Locale's alternative date representation.
`%EX`	Locale's alternative time representation.
`%Ey`	Offset from `%EC` (year only) in the locale's alternative representation.
`%EY`	Full alternative year representation.
`%Od`	Day of month using the locale's alternative numeric symbols.
`%Oe`	Day of month using the locale's alternative numeric symbols.
`%OH`	Hour (24-hour clock) using the locale's alternative numeric symbols.
`%OI`	Hour (12-hour clock) using the locale's alternative numeric symbols.
`%Om`	Month using the locale's alternative numeric symbols.
`%OM`	Minutes using the locale's alternative numeric symbols.
`%OS`	Seconds using the locale's alternative numeric symbols.
`%Ou`	Weekday as a number in the locale's alternative representation (Monday = 1).
`%OU`	Week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.
`%OV`	Week number of the year (Monday as the first day of the week, rules corresponding to `%V`), using the locale's alternative numeric symbols.
`%Ow`	Weekday as a number in the locale's alternative representation (Sunday = 0).
`%OW`	Week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
`%Oy`	Year (offset from `%C`) in alternative representation.

XSI *mmddhhmm***[[***cc***]***yy* **]**

Attempt to set the system date and time from the value given in the operand. This is only possible if the user has appropriate privileges and the system permits the setting of the system date and time. The first *mm* is the month (number); *dd* is the day (number); *hh* is the hour (number, 24-hour system); the second *mm* is the minute (number); *cc* is the century and is the first two digits of the year (this is optional); *yy* is the last two digits of the year and is optional. If century is not specified, then values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive. The current year is the default if *yy* is omitted.

Note: It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *date*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_TIME	Determine the format and contents of date and time strings written by *date*.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
TZ	Determine the timezone in which the time and date are written, unless the −**u** option is specified. If the *TZ* variable is unset or null and the −**u** is not specified, an unspecified system default timezone is used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When no formatting operand is specified, the output in the POSIX locale shall be equivalent to specifying:

```
date "+%a %b %e %H:%M:%S %Z %Y"
```

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The date was written successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Conversion specifiers are of unspecified format when not in the POSIX locale. Some of them can contain <newline>s in some locales, so it may be difficult to use the format shown in standard output for parsing the output of *date* in those locales.

The range of values for `%S` extends from 0 to 60 seconds to accommodate the occasional leap second.

Although certain of the conversion specifiers in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these fields may need to adjust the capitalization if the output is going to be used at the beginning of a sentence.

The date string formatting capabilities are intended for use in Gregorian-style calendars, possibly with a different starting year (or years). The `%x` and `%c` conversion specifications, however, are intended for local representation; these may be based on a different, non-Gregorian calendar.

The `%C` conversion specification was introduced to allow a fallback for the `%EC` (alternative year format base year); it can be viewed as the base of the current subdivision in the Gregorian calendar. The century number is calculated as the year divided by 100 and truncated to an integer; it should not be confused with the use of ordinal numbers for centuries (for example, ‘‘twenty-first century’’.) Both the `%Ey` and `%y` can then be viewed as the offset from `%EC` and `%C`, respectively.

The `E` and `O` modifiers modify the traditional conversion specifiers, so that they can always be used, even if the implementation (or the current locale) does not support the modifier.

The `E` modifier supports alternative date formats, such as the Japanese Emperor’s Era, as long as these are based on the Gregorian calendar system. Extending the `E` modifiers to other date elements may provide an implementation-defined extension capable of supporting other calendar systems, especially in combination with the `O` modifier.

The `O` modifier supports time and date formats using the locale’s alternative numerical symbols, such as Kanji or Hindi digits or ordinal number representation.

Non-European locales, whether they use Latin digits in computational items or not, often have local forms of the digits for use in date formats. This is not totally unknown even in Europe; a variant of dates uses Roman numerals for the months: the third day of September 1991 would be written as 3.IX.1991. In Japan, Kanji digits are regularly used for dates; in Arabic-speaking countries, Hindi digits are used. The `%d`, `%e`, `%H`, `%I`, `%m`, `%S`, `%U`, `%w`, `%W`, and `%y` conversion specifications always return the date and time field in Latin digits (that is, 0 to 9). The `%O` modifier was introduced to support the use for display purposes of non-Latin digits. In the *LC_TIME* category in *localedef*, the optional **alt_digits** keyword is intended for this purpose. As an example, assume the following (partial) *localedef* source:

```
alt_digits  "";"I";"II";"III";"IV";"V";"VI";"VII";"VIII" \
            "IX";"X";"XI";"XII"
d_fmt       "%e.%Om.%Y"
```

With the above date, the command:

```
date "+%x"
```

would yield 3.IX.1991. With the same **d_fmt**, but without the **alt_digits**, the command would yield 3.9.1991.

EXAMPLES

1. The following are input/output examples of *date* used at arbitrary times in the POSIX locale:

```
$ date
Tue Jun 26 09:58:10 PDT 1990
```

```
$ date "+DATE: %m/%d/%y%nTIME: %H:%M:%S"
DATE: 11/02/91
TIME: 13:36:16
```

```
$ date "+TIME: %r"
TIME: 01:36:32 PM
```

2. Examples for Denmark, where the default date and time format is `%a %d %b %Y %T %Z`:

```
$ LANG=da_DK.iso_8859−1 date
ons 02 okt 1991 15:03:32 CET
```

```
$ LANG=da_DK.iso_8859−1 \
    date "+DATO: %A den %e. %B %Y%nKLOKKEN: %H:%M:%S"
DATO: onsdag den 2. oktober 1991
KLOKKEN: 15:03:56
```

3. Examples for Germany, where the default date and time format is `%a %d.%h.%Y, %T %Z`:

```
$ LANG=De_DE.88591 date
Mi 02.Okt.1991, 15:01:21 MEZ
```

```
$ LANG=De_DE.88591 date "+DATUM: %A, %d. %B %Y%nZEIT: %H:%M:%S"
DATUM: Mittwoch, 02. Oktober 1991
ZEIT: 15:02:02
```

4. Examples for France, where the default date and time format is `%a %d %h %Y %Z %T`:

```
$ LANG=Fr_FR.88591 date
Mer 02 oct 1991 MET 15:03:32
```

```
$ LANG=Fr_FR.88591 date "+JOUR: %A %d %B %Y%nHEURE: %H:%M:%S"
JOUR: Mercredi 02 octobre 1991
HEURE: 15:03:56
```

RATIONALE

Some of the new options for formatting are from the ISO C standard. The **−u** option was introduced to allow portable access to Coordinated Universal Time (UTC). The string `"GMT0"` is allowed as an equivalent *TZ* value to be compatible with all of the systems using the BSD implementation, where this option originated.

The `%e` format conversion specifications (adopted from System V) was added because the ISO C standard conversion specifications did not provide any way to produce the historical default *date* output during the first nine days of any month.

There are two varieties of day and week numbering supported (in addition to any others created with the locale-dependent `%E` and `%O` modifier characters):

- The historical variety in which Sunday is the first day of the week and the weekdays preceding the first Sunday of the year are considered week 0. These are represented by `%w` and `%U`. A variant of this is `%W`, using Monday as the first day of the week, but still referring to week 0. This view of the calendar was retained because so many historical applications depend on it and the ISO C standard *strftime*( ) function, on which many *date* implementations are based, was defined in this way.

- The international standard, based on the ISO 8601:2000 standard where Monday is the first weekday and the algorithm for the first week number is more complex: If the week (Monday to Sunday) containing January 1 has four or more days in the new year, then it is week 1; otherwise, it is week 53 of the previous year, and the next week is week 1. These are represented by the new conversion specifications `%u` and `%V`, added as a result of international comments.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *printf*( ), *strftime*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

Changes are made for Year 2000 alignment.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The setting of system date and time is described, including how to interpret two-digit year values if a century is not given.
- The `%EX` modified conversion specification is added.

The Open Group Corrigendum U048/2 is applied, correcting the examples.

The DESCRIPTION is updated to refer to conversion specifications, instead of field descriptors for consistency with the *LC_TIME* category.

A clarification is made such that the current year is the default if the *yy* argument is omitted when setting the system date and time.

NAME

dd — convert and copy a file

SYNOPSIS

```
dd [operand ...]
```

DESCRIPTION

The *dd* utility shall copy the specified input file to the specified output file with possible conversions using specific input and output block sizes. It shall read the input one block at a time, using the specified input block size; it shall then process the block of data actually returned, which could be smaller than the requested block size. It shall apply any conversions that have been specified and write the resulting data to the output in blocks of the specified output block size. If the **bs**=*expr* operand is specified and no conversions other than **sync**, **noerror**, or **notrunc** are requested, the data returned from each input block shall be written as a separate output block; if the read returns less than a full block and the **sync** conversion is not specified, the resulting output block shall be the same size as the input block. If the **bs**=*expr* operand is not specified, or a conversion other than **sync**, **noerror**, or **notrunc** is requested, the input shall be processed and collected into full-sized output blocks until the end of the input is reached.

The processing order shall be as follows:

1. An input block is read.
2. If the input block is shorter than the specified input block size and the **sync** conversion is specified, null bytes shall be appended to the input data up to the specified size. (If either **block** or **unblock** is also specified, <space>s shall be appended instead of null bytes.) The remaining conversions and output shall include the pad characters as if they had been read from the input.
3. If the **bs**=*expr* operand is specified and no conversion other than **sync** or **noerror** is requested, the resulting data shall be written to the output as a single block, and the remaining steps are omitted.
4. If the **swab** conversion is specified, each pair of input data bytes shall be swapped. If there is an odd number of bytes in the input block, the last byte in the input record shall not be swapped.
5. Any remaining conversions (**block**, **unblock**, **lcase**, and **ucase**) shall be performed. These conversions shall operate on the input data independently of the input blocking; an input or output fixed-length record may span block boundaries.
6. The data resulting from input or conversion or both shall be aggregated into output blocks of the specified size. After the end of input is reached, any remaining output shall be written as a block without padding if **conv**=**sync** is not specified; thus, the final output block may be shorter than the output block size.

OPTIONS

None.

OPERANDS

All of the operands shall be processed before any input is read. The following operands shall be supported:

if=*file* Specify the input pathname; the default is standard input.

of=*file* Specify the output pathname; the default is standard output. If the **seek**=*expr* conversion is not also specified, the output file shall be truncated before the copy begins if an explicit **of**=*file* operand is specified, unless **conv**=**notrunc** is specified. |

If **seek**=*expr* is specified, but **conv**=**notrunc** is not, the effect of the copy shall be to preserve the blocks in the output file over which *dd* seeks, but no other portion of the output file shall be preserved. (If the size of the seek plus the size of the input file is less than the previous size of the output file, the output file shall be shortened by the copy.)

ibs=*expr* Specify the input block size, in bytes, by *expr* (default is 512).

obs=*expr* Specify the output block size, in bytes, by *expr* (default is 512).

bs=*expr* Set both input and output block sizes to *expr* bytes, superseding **ibs**= and **obs**=. If no conversion other than **sync**, **noerror**, and **notrunc** is specified, each input block shall be copied to the output as a single block without aggregating short blocks.

cbs=*expr* Specify the conversion block size for **block** and **unblock** in bytes by *expr* (default is zero). If **cbs**= is omitted or given a value of zero, using **block** or **unblock** produces unspecified results.

XSI The application shall ensure that this operand is also specified if the **conv**= operand is specified with a value of **ascii**, **ebcdic**, or **ibm**. For a **conv**= operand with an **ascii** value, the input is handled as described for the **unblock** value, except that characters are converted to ASCII before any trailing <space>s are deleted. For **conv**= operands with **ebcdic** or **ibm** values, the input is handled as described for the **block** value except that the characters are converted to EBCDIC or IBM EBCDIC, respectively, after any trailing <space>s are added.

skip=*n* Skip *n* input blocks (using the specified input block size) before starting to copy. On seekable files, the implementation shall read the blocks or seek past them; on non-seekable files, the blocks shall be read and the data shall be discarded.

seek=*n* Skip *n* blocks (using the specified output block size) from beginning of the output file before copying. On non-seekable files, existing blocks shall be read and space from the current end-of-file to the specified offset, if any, filled with null bytes; on seekable files, the implementation shall seek to the specified offset or read the blocks as described for non-seekable files.

count=*n* Copy only *n* input blocks.

conv=*value*[,*value* . . .]

Where *value*s are comma-separated symbols from the following list:

XSI **ascii** Convert EBCDIC to ASCII; see Table 4-6 (on page 2506).

XSI **ebcdic** Convert ASCII to EBCDIC; see Table 4-6 (on page 2506).

XSI **ibm** Convert ASCII to a different EBCDIC set; see Table 4-7 (on page 2507).

The **ascii**, **ebcdic**, and **ibm** values are mutually-exclusive.

block Treat the input as a sequence of <newline>-terminated or end-of-file-terminated variable-length records independent of the input block boundaries. Each record shall be converted to a record with a fixed length specified by the conversion block size. Any <newline> shall be removed from the input line; <space>s shall be appended to lines that are shorter than their conversion block size to fill the block. Lines that are longer than the conversion block size shall be truncated to the largest number of characters that fit into that size; the number of truncated lines shall be reported (see the STDERR section).

The **block** and **unblock** values are mutually-exclusive.

unblock	Convert fixed-length records to variable length. Read a number of bytes equal to the conversion block size (or the number of bytes remaining in the input, if less than the conversion block size), delete all trailing <space>s, and append a <newline>.
lcase	Map uppercase characters specified by the *LC_CTYPE* keyword **tolower** to the corresponding lowercase character. Characters for which no mapping is specified shall not be modified by this conversion. The **lcase** and **ucase** symbols are mutually-exclusive.
ucase	Map lowercase characters specified by the *LC_CTYPE* keyword **toupper** to the corresponding uppercase character. Characters for which no mapping is specified shall not be modified by this conversion.
swab	Swap every pair of input bytes.
noerror	Do not stop processing on an input error. When an input error occurs, a diagnostic message shall be written on standard error, followed by the current input and output block counts in the same format as used at completion (see the STDERR section). If the **sync** conversion is specified, the missing input shall be replaced with null bytes and processed normally; otherwise, the input block shall be omitted from the output.
notrunc	Do not truncate the output file. Preserve blocks in the output file not explicitly written by this invocation of the *dd* utility. (See also the preceding **of**=*file* operand.)
sync	Pad every input block to the size of the **ibs**= buffer, appending null bytes. (If either **block** or **unblock** is also specified, append <space>s, rather than null bytes.)

The behavior is unspecified if operands other than **conv**= are specified more than once.

For the **bs**=, **cbs**=, **ibs**=, and **obs**= operands, the application shall supply an expression specifying a size in bytes. The expression, *expr*, can be:

1. A positive decimal number
2. A positive decimal number followed by *k*, specifying multiplication by 1 024
3. A positive decimal number followed by *b*, specifying multiplication by 512
4. Two or more positive decimal numbers (with or without *k* or *b*) separated by *x*, specifying the product of the indicated values

All of the operands are processed before any input is read.

XSI The following two tables display the octal number character values used for the **ascii** and **ebcdic** conversions (first table) and for the **ibm** conversion (second table). In both tables, the ASCII values are the row and column headers and the EBCDIC values are found at their intersections. For example, ASCII 0012 (LF) is the second row, third column, yielding 0045 in EBCDIC. The inverted tables (for EBCDIC to ASCII conversion) are not shown, but are in one-to-one correspondence with these tables. The differences between the two tables are highlighted by small boxes drawn around five entries.

Table 4-6 ASCII to EBCDIC Conversion

	0	1	2	3	4	5	6	7
0000	0000 NUL	0001 SOH	0002 STX	0003 ETX	0067 EOT	0055 ENQ	0056 ACK	0057 BEL
0010	0026 BS	0005 HT	0045 LF	0013 VT	0014 FF	0015 CR	0016 SO	0017 SI
0020	0020 DLE	0021 DC1	0022 DC2	0023 DC3	0074 DC4	0075 NAK	0062 SYN	0046 ETB
0030	0030 CAN	0031 EM	0077 SUB	0047 ESC	0034 IFS	0035 IGS	0036 IRS	0037 ITB
0040	0100 Sp	0132 !	0177 "	0173 #	0133 $	0154 %	0120 &	0175 '
0050	0115 (	0135)	0134 *	0116 +	0153 ,	0140 -	0113 .	0141 /
0060	0360 0	0361 1	0362 2	0363 3	0364 4	0365 5	0366 6	0367 7
0070	0370 8	0371 9	0172 :	0136 ;	0114 <	0176 =	0156 >	0157 ?
0100	0174 @	0301 A	0302 B	0303 C	0304 D	0305 E	0306 F	0307 G
0110	0310 H	0311 I	0321 J	0322 K	0323 L	0324 M	0325 N	0326 O
0120	0327 P	0330 Q	0331 R	0342 S	0343 T	0344 U	0345 V	0346 W
0130	0347 X	0350 Y	0351 Z	0255 [	0340 \	0275]	0232	0155 _
0140	0171 `	0201 a	0202 b	0203 c	0204 d	0205 e	0206 f	0207 g
0150	0210 h	0211 i	0221 j	0222 k	0223 l	0224 m	0225 n	0226 o
0160	0227 p	0230 q	0231 r	0242 s	0243 t	0244 u	0245 v	0246 w
0170	0247 x	0250 y	0251 z	0300 {	0117 |	0320 }	0137 ¬	0007 DEL
0200	0040 DS	0041 SOS	0042 FS	0043 WUS	0044 BYP	0025 NL	0006 RNL	0027 POC
0210	0050 SA	0051 SFE	0052 SM	0053 CSP	0054 MFA	0011 SPS	0012 RPT	0033 CU1
0220	0060	0061	0032 UBS	0063 IR	0064 PP	0065 TRN	0066 NBS	0010 GE
0230	0070 SBS	0071 IT	0072 RFF	0073 CU3	0004 SEL	0024 RES	0076	0341
0240	0101	0102	0103	0104	0105	0106	0107	0110
0250	0111	0121	0122	0123	0124	0125	0126	0127
0260	0130	0131	0142	0143	0144	0145	0146	0147
0270	0150	0151	0160	0161	0162	0163	0164	0165
0300	0166	0167	0170	0200	0212	0213	0214	0215
0310	0216	0217	0220	0152 ¦	0233	0234	0235	0236
0320	0237	0240	0252	0253	0254	0112 ¢	0256	0257
0330	0260	0261	0262	0263	0264	0265	0266	0267
0340	0270	0271	0272	0273	0274	0241	0276	0277
0350	0312	0313	0314 ʃ	0315	0316 Ɏ	0317	0332	0333
0360	0334	0335	0336	0337	0352	0353	0354 Ⅎ	0355
0370	0356	0357	0372 |	0373	0374	0375	0376	0377 EO

Table 4-7 ASCII to IBM EBCDIC Conversion

	0	1	2	3	4	5	6	7
0000	0000 NUL	0001 SOH	0002 STX	0003 ETX	0067 EOT	0055 ENQ	0056 ACK	0057 BEL
0010	0026 BS	0005 HT	0045 LF	0013 VT	0014 FF	0015 CR	0016 SO	0017 SI
0020	0020 DLE	0021 DC1	0022 DC2	0023 DC3	0074 DC4	0075 NAK	0062 SYN	0046 ETB
0030	0030 CAN	0031 EM	0077 SUB	0047 ESC	0034 IFS	0035 IGS	0036 IRS	0037 ITB
0040	0100 Sp	0132 !	0177 "	0173 #	0133 $	0154 %	0120 &	0175 '
0050	0115 (	0135)	0134 *	0116 +	0153 ,	0140 -	0113 .	0141 /
0060	0360 0	0361 1	0362 2	0363 3	0364 4	0365 5	0366 6	0367 7
0070	0370 8	0371 9	0172 :	0136 ;	0114 <	0176 =	0156 >	0157 ?
0100	0174 @	0301 A	0302 B	0303 C	0304 D	0305 E	0306 F	0307 G
0110	0310 H	0311 I	0321 J	0322 K	0323 L	0324 M	0325 N	0326 O
0120	0327 P	0330 Q	0331 R	0342 S	0343 T	0344 U	0345 V	0346 W
0130	0347 X	0350 Y	0351 Z	0255 [	0340 \	0275]	0137 ¬	0155 _
0140	0171 `	0201 a	0202 b	0203 c	0204 d	0205 e	0206 f	0207 g
0150	0210 h	0211 i	0221 j	0222 k	0223 l	0224 m	0225 n	0226 o
0160	0227 p	0230 q	0231 r	0242 s	0243 t	0244 u	0245 v	0246 w
0170	0247 x	0250 y	0251 z	0300 {	0117 |	0320 }	0241	0007 DEL
0200	0040 DS	0041 SOS	0042 FS	0043 WUS	0044 BYP	0025 NL	0006 RNL	0027 POC
0210	0050 SA	0051 SFE	0052 SM	0053 CSP	0054 MFA	0011 SPS	0012 RPT	0033 CU1
0220	0060	0061	0032 UBS	0063 IR	0064 PP	0065 TRN	0066 NBS	0010 GE
0230	0070 SBS	0071 IT	0072 RFF	0073 CU3	0004 SEL	0024 RES	0076	0341
0240	0101	0102	0103	0104	0105	0106	0107	0110
0250	0111	0121	0122	0123	0124	0125	0126	0127
0260	0130	0131	0142	0143	0144	0145	0146	0147
0270	0150	0151	0160	0161	0162	0163	0164	0165
0300	0166	0167	0170	0200	0212	0213	0214	0215
0310	0216	0217	0220	0232	0233	0234	0235	0236
0320	0237	0240	0252	0253	0254	0255 [	0256	0257
0330	0260	0261	0262	0263	0264	0265	0266	0267
0340	0270	0271	0272	0273	0274	0275]	0276	0277
0350	0312	0313	0314 ⑀	0315	0316 ⑂	0317	0332	0333
0360	0334	0335	0336	0337	0352	0353	0354 ⑁	0355
0370	0356	0357	0372 |	0373	0374	0375	0376	0377 EO

STDIN

If no **if**= operand is specified, the standard input shall be used. See the INPUT FILES section.

INPUT FILES

The input file can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *dd*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the classification of characters as uppercase or lowercase, and the mapping of characters from one case to the other.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

For SIGINT, the *dd* utility shall interrupt its current processing, write status information to standard error, and exit as though terminated by SIGINT. It shall take the standard action for all other signals; see the ASYNCHRONOUS EVENTS section in Section 1.11 (on page 2221).

STDOUT

If no **of**= operand is specified, the standard output shall be used. The nature of the output depends on the operands selected.

STDERR

On completion, *dd* shall write the number of input and output blocks to standard error. In the POSIX locale the following formats shall be used:

```
"%u+%u records in\n", <number of whole input blocks>,
    <number of partial input blocks>
```

```
"%u+%u records out\n", <number of whole output blocks>,
    <number of partial output blocks>
```

A partial input block is one for which *read*( ) returned less than the input block size. A partial output block is one that was written with fewer bytes than specified by the output block size.

In addition, when there is at least one truncated block, the number of truncated blocks shall be written to standard error. In the POSIX locale, the format shall be:

```
"%u truncated %s\n", <number of truncated blocks>, "record" (if
    <number of truncated blocks> is one) "records" (otherwise)
```

Diagnostic messages may also be written to standard error.

OUTPUT FILES

If the **of**= operand is used, the output shall be the same as described in the STDOUT section.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The input file was copied successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If an input error is detected and the **noerror** conversion has not been specified, any partial output block shall be written to the output file, a diagnostic message shall be written, and the copy operation shall be discontinued. If some other error is detected, a diagnostic message shall be written and the copy operation shall be discontinued.

APPLICATION USAGE

The input and output block size can be specified to take advantage of raw physical I/O.

There are many different versions of the EBCDIC codesets. The ASCII and EBCDIC conversions specified for the *dd* utility perform conversions for the version specified by the tables.

EXAMPLES

The following command:

```
dd if=/dev/rmt0h  of=/dev/rmt1h
```

copies from tape drive 0 to tape drive 1, using a common historical device naming convention.

The following command:

```
dd ibs=10  skip=1
```

strips the first 10 bytes from standard input.

This example reads an EBCDIC tape blocked ten 80-byte EBCDIC card images per block into the ASCII file **x**:

```
dd if=/dev/tape of=x ibs=800 cbs=80 conv=ascii,lcase
```

RATIONALE

The OPTIONS section is listed as ‘‘None’’ because there are no options recognized by historical *dd* utilities. Certainly, many of the operands could have been designed to use the Utility Syntax Guidelines, which would have resulted in the classic hyphenated option letters. In this version of this volume of IEEE Std 1003.1-200x, *dd* retains its curious JCL-like syntax due to the large number of applications that depend on the historical implementation.

A suggested implementation technique for **conv**=**noerror**,**sync** is to zero (or <space>-fill, if **block**ing or **unblock**ing) the input buffer before each read and to write the contents of the input buffer to the output even after an error. In this manner, any data transferred to the input buffer before the error was detected is preserved. Another point is that a failed read on a regular file or a disk generally does not increment the file offset, and *dd* must then seek past the block on which the error occurred; otherwise, the input error occurs repetitively. When the input is a magnetic tape, however, the tape normally has passed the block containing the error when the error is reported, and thus no seek is necessary.

The default **ibs**= and **obs**= sizes are specified as 512 bytes because there are historical (largely portable) scripts that assume these values. If they were left unspecified, unusual results could occur if an implementation chose an odd block size.

Historical implementations of *dd* used *creat*( ) when processing **of**=*file*. This makes the **seek**= operand unusable except on special files. The **conv**=**notrunc** feature was added because more recent BSD-based implementations use *open*( ) (without O_TRUNC) instead of *creat*( ), but they fail to delete output file contents after the data copied.

The *w* multiplier (historically meaning *word*), is used in System V to mean 2 and in 4.2 BSD to mean 4. Since *word* is inherently non-portable, its use is not supported by this volume of IEEE Std 1003.1-200x.

Standard EBCDIC does not have the characters '[' and ']'. The values used in the table are taken from a common print train that does contain them. Other than those characters, the print train values are not filled in, but appear to provide some of the motivation for the historical choice of translations reflected here.

The Standard EBCDIC table provides a 1:1 translation for all 256 bytes.

The IBM EBCDIC table does not provide such a translation. The marked cells in the tables differ in such a way that:

1. EBCDIC 0112 ('¢') and 0152 (broken pipe) do not appear in the table.
2. EBCDIC 0137 ('¬') translates to/from ASCII 0236 ('^'). In the standard table, EBCDIC 0232 (no graphic) is used.
3. EBCDIC 0241 ('˜') translates to/from ASCII 0176 ('˜'). In the standard table, EBCDIC 0137 ('¬') is used.
4. 0255 ('[') and 0275 (']') appear twice, once in the same place as for the standard table and once in place of 0112 ('¢') and 0241 ('˜').

In net result:

EBCDIC 0275 (']') displaced EBCDIC 0241 ('˜') in cell 0345.

That displaced EBCDIC 0137 ('¬') in cell 0176.

That displaced EBCDIC 0232 (no graphic) in cell 0136.

That replaced EBCDIC 0152 (broken pipe) in cell 0313.

EBCDIC 0255 ('[') replaced EBCDIC 0112 ('¢').

This translation, however, reflects historical practice that (ASCII) '˜' and '¬' were often mapped to each other, as were '[' and '¢'; and ']' and (EBCDIC) '˜'.

The **cbs** operand is required if any of the **ascii**, **ebcdic**, or **ibm** operands are specified. For the **ascii** operand, the input is handled as described for the **unblock** operand except that characters are converted to ASCII before the trailing <space>s are deleted. For the **ebcdic** and **ibm** operands, the input is handled as described for the **block** operand except that the characters are converted to EBCDIC or IBM EBCDIC after the trailing <space>s are added.

The **block** and **unblock** keywords are from historical BSD practice.

The consistent use of the word **record** in standard error messages matches most historical practice. An earlier version of System V used **block**, but this has been updated in more recent releases.

Early proposals only allowed two numbers separated by **x** to be used in a product when specifying **bs**=, **cbs**=, **ibs**=, and **obs**= sizes. This was changed to reflect the historical practice of allowing multiple numbers in the product as provided by Version 7 and all releases of System V and BSD.

A change to the *swab* conversion is required to match historical practice and is the result of IEEE PASC Interpretation 1003.2 #03 and #04, submitted for the ISO POSIX-2: 1993 standard.

A change to the handling of SIGINT is required to match historical practice and is the result of IEEE PASC Interpretation 1003.2 #06 submitted for the ISO POSIX-2: 1993 standard.

FUTURE DIRECTIONS

None.

SEE ALSO

sed, *tr*

CHANGE HISTORY

First released in Issue 2.

Issue 5

The second paragraph of the **cbs**= description is reworded and marked EX.

FUTURE DIRECTIONS section added.

Issue 6

Changes are made to *swab* conversion and SIGINT handling to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ''must'' for application requirements.

IEEE PASC Interpretation 1003.2 #209 is applied, clarifying the interaction between *dd* **of**=*file* and **conv**=**notrunc**.

NAME

delta — make a delta (change) to an SCCS file (**DEVELOPMENT**)

SYNOPSIS

XSI

```
delta [−nps][−g list][−m mrlist][−r SID][−y[comment]] file...
```

DESCRIPTION

The *delta* utility shall be used to permanently introduce into the named SCCS files changes that were made to the files retrieved by *get* (called the *g-files*, or generated files).

OPTIONS

The *delta* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the **−y** option has an optional option-argument. This optional option-argument shall not be presented as a separate argument.

The following options shall be supported:

−r *SID*	Uniquely identify which delta is to be made to the SCCS file. The use of this option shall be necessary only if two or more outstanding *get* commands for editing (*get* **−e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **−r** option can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* utility; see *get* (on page 2675).
−s	Suppress the report to standard output of the activity associated with each *file*. See the STDOUT section.
−n	Specify retention of the edited *g-file* (normally removed at completion of delta processing).
−g *list*	Specify a *list*, (see *get* (on page 2675) for the definition of *list*) of deltas that shall be ignored when the file is accessed at the change level (SID) created by this delta.
−m *mrlist*	Specify a modification request (MR) number that the application shall supply as the reason for creating the new delta. This shall be used if the SCCS file has the **v** flag set; see *admin* (on page 2328). If **−m** is not used and `'−'` is not specified as a file argument, and the standard input is a terminal, the prompt described in the STDOUT section shall be written to standard output before the standard input is read; if the standard input is not a terminal, no prompt shall be issued. MRs in a list shall be separated by <blank>s or escaped <newline>s. An unescaped <newline> shall terminate the MR list. The escape character is <backslash>. If the **v** flag has a value, it shall be taken to be the name of a program which validates the correctness of the MR numbers. If a non-zero exit status is returned from the MR number validation program, the *delta* utility shall terminate. (It is assumed that the MR numbers were not all valid.)
−y[*comment***]**	Describe the reason for making the delta. The *comment* shall be an arbitrary group of lines that would meet the definition of a text file. Implementations shall support *comment*s from zero to 512 bytes and may support longer values. A null string (specified as either **−y**, **−y" "**, or in response to a prompt for a comment) shall be considered a valid *comment*.

If −**y** is not specified and '−' is not specified as a file argument, and the standard input is a terminal, the prompt described in the STDOUT section shall be written to standard output before the standard input is read; if the standard input is not a terminal, no prompt shall be issued. An unescaped <newline> shall terminate the comment text. The escape character is <backslash>.

The −**y** option shall be required if the *file* operand is specified as '−'.

−**p** Write (to standard output) the SCCS file differences before and after the delta is applied in *diff* format; see *diff* (on page 2520).

OPERANDS

The following operand shall be supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, the *delta* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files shall be silently ignored.

If exactly one *file* operand appears, and it is '−', the standard input shall be read; each line of the standard input shall be taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only in the following cases:

- To read an *mrlist* or a *comment* (see the −**m** and −**y** options).
- A *file* operand shall be specified as '−'. In this case, the −**y** option must be used to specify the comment, and if the SCCS file has the **v** flag set, the −**m** option must also be used to specify the MR list.

INPUT FILES

Input files shall be text files whose data is to be included in the SCCS files. If the first character of any line of an input file is <SOH> in the POSIX locale, the results are unspecified. If this file contains more than 99 999 lines, the number of lines recorded in the header for this file shall be 99 999 for this delta.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *delta*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES.*

TZ	Determine the timezone in which the time and date are written in the SCCS file. If the *TZ* variable is unset or NULL, an unspecified system default timezone is used.

ASYNCHRONOUS EVENTS

If SIGINT is caught, temporary files shall be cleaned up and *delta* shall exit with a non-zero exit code. The standard action shall be taken for all other signals; see Section 1.11 (on page 2221).

STDOUT

The standard output shall be used only for the following messages in the POSIX locale:

- Prompts (see the **–m** and **–y** options) in the following formats:

  `"MRs? "`

  `"comments? "`

  The MR prompt, if written, shall always precede the comments prompt.

- A report of each *file*'s activities (unless the **–s** option is specified) in the following format:

```
"%s\n%d inserted\n%d deleted\n%d unchanged\n", <New SID>,
    <number of lines inserted>, <number of lines deleted>,
    <number of lines unchanged>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Any SCCS files updated shall be files of an unspecified format.

EXTENDED DESCRIPTION

System Date and Time

When a delta is added to an SCCS file, the system date and time shall be recorded for the new delta. If a *get* is performed using an SCCS file with a date recorded apparently in the future, the behavior is unspecified.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Problems can arise if the system date and time have been modified (for example, put forward and then back again, or unsynchronized clocks across a network) and can also arise when different values of the *TZ* environment variable are used.

Problems of a similar nature can also arise for the operation of the *get* utility, which records the date and time in the file body.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

admin, *diff*, *get*, *prs*, *rmdel*

CHANGE HISTORY

First released in Issue 2.

Issue 5

The output format description in the STDOUT section is corrected.

Issue 6

The APPLICATION USAGE section is added.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

The Open Group Base Resolution bwg2001-007 is applied as follows:

- The use of '−' as a file argument is clarified.
- The use of STDIN is added.
- The ASYNCHRONOUS EVENTS section is updated to remove the implicit requirement that implementations re-signal themselves when catching a normally fatal signal.
- New text is added to the INPUT FILES section warning that the maximum lines recorded in the file is 99 999.

New text is added to the EXTENDED DESCRIPTION and APPLICAION USAGE sections regarding how the system date and time may be taken into account, and the *TZ* environment variable is added to the ENVIRONMENT VARIABLES section as per The Open Group Base Resolution bwg2001-007.

NAME

df — report free disk space

SYNOPSIS

`df [−k][−P|−t][file...]`

DESCRIPTION

The *df* utility shall write the amount of available space and file slots for file systems on which the invoking user has appropriate read access. File systems shall be specified by the *file* operands; when none are specified, information shall be written for all file systems. The format of the default output from *df* is unspecified, but all space figures are reported in 512-byte units, unless the **−k** option is specified. This output shall contain at least the file system names, amount of available space on each of these file systems, and the number of free file slots, or *inodes*, available; when **−t** is specified, the output shall contain the total allocated space as well.

OPTIONS

The *df* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−k	Use 1024-byte units, instead of the default 512-byte units, when writing space figures.
−P	Produce output in the format described in the STDOUT section.
−t	Include total allocated-space figures in the output.

OPERANDS

The following operand shall be supported:

file	A pathname of a file within the hierarchy of the desired file system. If a file other than a FIFO, a regular file, a directory or a special file representing the device containing the file system (for example, **/dev/dsk/0s1**) is specified, the results are unspecified. Otherwise, *df* shall write the amount of free space in the file system containing the specified *file* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *df*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

When both the −**k** and −**P** options are specified, the following header line shall be written (in the POSIX locale):

```
"Filesystem 1024-blocks Used Available Capacity Mounted on\n"
```

When the −**P** option is specified without the −**k** option, the following header line shall be written (in the POSIX locale):

```
"Filesystem 512-blocks Used Available Capacity Mounted on\n"
```

The implementation may adjust the spacing of the header line and the individual data lines so that the information is presented in orderly columns.

The remaining output with −**P** shall consist of one line of information for each specified file system. These lines shall be formatted as follows:

```
"%s %d %d %d %d%% %s\n", <file system name>, <total space>,
    <space used>, <space free>, <percentage used>,
    <file system root>
```

In the following list, all quantities expressed in 512-byte units (1 024-byte when −**k** is specified) shall be rounded up to the next higher unit. The fields are:

<file system name>
The name of the file system, in an implementation-defined format.

<total space> The total size of the file system in 512-byte units. The exact meaning of this figure is implementation-defined, but should include *<space used>*, *<space free>*, plus any space reserved by the system not normally available to a user.

<space used> The total amount of space allocated to existing files in the file system, in 512-byte units.

<space free> The total amount of space available within the file system for the creation of new files by unprivileged users, in 512-byte units. When this figure is less than or equal to zero, it shall not be possible to create any new files on the file system without first deleting others, unless the process has appropriate privileges. The figure written may be less than zero.

<percentage used>
The percentage of the normally available space that is currently allocated to all files on the file system. This shall be calculated using the fraction:

```
<space used>/( <space used>+ <space free>)
```

expressed as a percentage. This percentage may be greater than 100 if *<space free>* is less than zero. The percentage value shall be expressed as a positive integer, with any fractional result causing it to be rounded to the next highest integer.

<file system root>
The directory below which the file system hierarchy appears.

XSI The output format is unspecified when **−t** is used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

On most systems, the ‘‘name of the file system, in an implementation-defined format’’ is the special file on which the file system is mounted.

On large file systems, the calculation specified for percentage used can create huge rounding errors.

EXAMPLES

1. The following example writes portable information about the **/usr** file system:

```
df −P /usr
```

2. Assuming that **/usr/src** is part of the **/usr** file system, the following produces the same output as the previous example:

```
df −P /usr/src
```

RATIONALE

The behavior of *df* with the **−P** option is the default action of the 4.2 BSD *df* utility. The uppercase **−P** was selected to avoid collision with a known industry extension using **−p**.

Historical *df* implementations vary considerably in their default output. It was therefore necessary to describe the default output in a loose manner to accommodate all known historical implementations and to add a portable option (**−P**) to provide information in a portable format.

The use of 512-byte units is historical practice and maintains compatibility with *ls* and other utilities in this volume of IEEE Std 1003.1-200x. This does not mandate that the file system itself be based on 512-byte blocks. The **−k** option was added as a compromise measure. It was agreed by the standard developers that 512 bytes was the best default unit because of its complete historical consistency on System V (*versus* the mixed 512/1 024-byte usage on BSD systems), and that a **−k** option to switch to 1 024-byte units was a good compromise. Users who prefer the more logical 1 024-byte quantity can easily alias *df* to *df* **−k** without breaking many historical scripts relying on the 512-byte units.

It was suggested that *df* and the various related utilities be modified to access a *BLOCKSIZE* environment variable to achieve consistency and user acceptance. Since this is not historical practice on any system, it is left as a possible area for system extensions and will be re-evaluated in a future version if it is widely implemented.

FUTURE DIRECTIONS

None.

SEE ALSO

find

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

NAME

diff — compare two files

SYNOPSIS

`diff [−c| −e| −f| −C n][−br] file1 file2`

DESCRIPTION

The *diff* utility shall compare the contents of *file1* and *file2* and write to standard output a list of changes necessary to convert *file1* into *file2*. This list should be minimal. No output shall be produced if the files are identical.

OPTIONS

The *diff* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−b	Cause any amount of white space at the end of a line to be treated as a single <newline> (that is, the white-space characters preceding the <newline> are ignored) and other strings of white-space characters, not including <newline>s, to compare equal.
−c	Produce output in a form that provides three lines of context.
−C n	Produce output in a form that provides *n* lines of context (where *n* shall be interpreted as a positive decimal integer).
−e	Produce output in a form suitable as input for the *ed* utility, which can then be used to convert *file1* into *file2*.
−f	Produce output in an alternative form, similar in format to **−e**, but not intended to be suitable as input for the *ed* utility, and in the opposite order.
−r	Apply *diff* recursively to files and directories of the same name when *file1* and *file2* are both directories.

OPERANDS

The following operands shall be supported:

file1, *file2*	A pathname of a file to be compared. If either the *file1* or *file2* operand is `'−'`, the standard input shall be used in its place.

If both *file1* and *file2* are directories, *diff* shall not compare block special files, character special files, or FIFO special files to any files and shall not compare regular files to directories. Further details are as specified in **Diff Directory Comparison Format** (on page 2521). The behavior of *diff* on other file types is implementation-defined when found in directories.

If only one of *file1* and *file2* is a directory, *diff* shall be applied to the non-directory file and the file contained in the directory file with a filename that is the same as the last component of the non-directory file.

STDIN

The standard input shall be used only if one of the *file1* or *file2* operands references standard input. See the INPUT FILES section.

INPUT FILES

The input files may be of any type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *diff*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
LC_TIME	Determine the locale for affecting the format of file timestamps written with the −**C** and −**c** options.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
TZ	Determine the timezone used for calculating file timestamps written with the −**C** and −**c** options. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Diff Directory Comparison Format

If both *file1* and *file2* are directories, the following output formats shall be used.

In the POSIX locale, each file that is present in only one directory shall be reported using the following format:

```
"Only in %s: %s\n", <directory pathname>, <filename>
```

In the POSIX locale, subdirectories that are common to the two directories may be reported with the following format:

```
"Common subdirectories: %s and %s\n", <directory1 pathname>,
    <directory2 pathname>
```

For each file common to the two directories if the two files are not to be compared, the following format shall be used in the POSIX locale:

```
"File %s is a %s while file %s is a %s\n", <directory1 pathname>,
    <file type of directory1 pathname>, <directory2 pathname>,
    <file type of directory2 pathname>
```

For each file common to the two directories, if the files are compared and are identical, no output shall be written. If the two files differ, the following format is written:

```
"diff %s %s %s\n", <diff_options>, <filename1>, <filename2>
```

where <*diff_options*> are the options as specified on the command line.

All directory pathnames listed in this section shall be relative to the original command line arguments. All other names of files listed in this section shall be filenames (pathname components).

Diff Binary Output Format

In the POSIX locale, if one or both of the files being compared are not text files, an unspecified format shall be used that contains the pathnames of two files being compared and the string `"differ"`.

If both files being compared are text files, depending on the options specified, one of the following formats shall be used to write the differences.

Diff Default Output Format

The default (without **−e**, **−f**, **−c**, or **−C** options) *diff* utility output shall contain lines of these forms:

`"%da%d\n", <num1>, <num2>`

`"%da%d,%d\n", <num1>, <num2>, <num3>`

`"%dd%d\n", <num1>, <num2>`

`"%d,%dd%d\n", <num1>, <num2>, <num3>`

`"%dc%d\n", <num1>, <num2>`

`"%d,%dc%d\n", <num1>, <num2>, <num3>`

`"%dc%d,%d\n", <num1>, <num2>, <num3>`

`"%d,%dc%d,%d\n", <num1>, <num2>, <num3>, <num4>`

These lines resemble *ed* subcommands to convert *file1* into *file2*. The line numbers before the action letters shall pertain to *file1*; those after shall pertain to *file2*. Thus, by exchanging *a* for *d* and reading the line in reverse order, one can also determine how to convert *file2* into *file1*. As in *ed*, identical pairs (where *num1*= *num2*) are abbreviated as a single number.

Following each of these lines, *diff* shall write to standard output all lines affected in the first file using the format:

`"<Δ%s", <line>`

and all lines affected in the second file using the format:

`">Δ%s", <line>`

If there are lines affected in both *file1* and *file2* (as with the **c** subcommand), the changes are separated with a line consisting of three hyphens:

`"−−−\n"`

Diff −e Output Format

With the −**e** option, a script shall be produced that shall, when provided as input to *ed*, along with an appended **w** (write) command, convert *file1* into *file2*. Only the **a** (append), **c** (change), **d** (delete), **i** (insert), and **s** (substitute) commands of *ed* shall be used in this script. Text lines, except those consisting of the single character period (`'.'`), shall be output as they appear in the file.

Diff −f Output Format

With the −**f** option, an alternative format of script shall be produced. It is similar to that produced by −**e**, with the following differences:

1. It is expressed in reverse sequence; the output of −**e** orders changes from the end of the file to the beginning; the −**f** from beginning to end.
2. The command form <*lines*> <*command-letter*> used by −**e** is reversed. For example, 10*c* with −**e** would be *c*10 with −**f**.
3. The form used for ranges of line numbers is <space>-separated, rather than comma-separated.

Diff −c or −C Output Format

With the −**c** or −**C** option, the output format shall consist of affected lines along with surrounding lines of context. The affected lines shall show which ones need to be deleted or changed in *file1*, and those added from *file2*. With the −**c** option, three lines of context, if available, shall be written before and after the affected lines. With the −**C** option, the user can specify how many lines of context are written. The exact format follows.

The name and last modification time of each file shall be output in the following format:

```
"*** %s %s\n", file1, <file1 timestamp>
"--- %s %s\n", file2, <file2 timestamp>
```

Each <*file*> field shall be the pathname of the corresponding file being compared. The pathname written for standard input is unspecified.

In the POSIX locale, each <*timestamp*> field shall be equivalent to the output from the following command:

```
date "+%a %b %e %T %Y"
```

without the trailing <newline>, executed at the time of last modification of the corresponding file (or the current time, if the file is standard input).

Then, the following output formats shall be applied for every set of changes.

First, a line shall be written in the following format:

```
"***************\n"
```

Next, the range of lines in *file1* shall be written in the following format:

```
"*** %d,%d ****\n", <beginning line number>, <ending line number>
```

Next, the affected lines along with lines of context (unaffected lines) shall be written. Unaffected lines shall be written in the following format:

```
"ΔΔ%s", <unaffected_line>
```

Deleted lines shall be written as:

```
"−∆%s", <deleted_line>
```

Changed lines shall be written as:

```
"!∆%s", <changed_line>
```

Next, the range of lines in *file2* shall be written in the following format:

```
"−−− %d,%d −−−−\n", <beginning line number>, <ending line number>
```

Then, lines of context and changed lines shall be written as described in the previous formats. Lines added from *file2* shall be written in the following format:

```
"+∆%s", <added_line>
```

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	No differences were found.
1	Differences were found.
>1	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If lines at the end of a file are changed and other lines are added, *diff* output may show this as a delete and add, as a change, or as a change and add; *diff* is not expected to know which happened and users should not care about the difference in output as long as it clearly shows the differences between the files.

EXAMPLES

If **dir1** is a directory containing a directory named **x**, **dir2** is a directory containing a directory named **x**, **dir1/x** and **dir2/x** both contain files named **date.out**, and **dir2/x** contains a file named **y**, the command:

```
diff −r dir1 dir2
```

could produce output similar to:

```
Common subdirectories: dir1/x and dir2/x
Only in dir2/x: y
diff −r dir1/x/date.out dir2/x/date.out
1c1
< Mon Jul  2 13:12:16 PDT 1990
−−−
> Tue Jun 19 21:41:39 PDT 1990
```

RATIONALE

The **–h** option was omitted because it was insufficiently specified and does not add to applications portability.

Historical implementations employ algorithms that do not always produce a minimum list of differences; the current language about making every effort is the best this volume of IEEE Std 1003.1-200x can do, as there is no metric that could be employed to judge the quality of implementations against any and all file contents. The statement ‘‘This list should be minimal’’ clearly implies that implementations are not expected to provide the following output when comparing two 100-line files that differ in only one character on a single line:

```
1,100c1,100
all 100 lines from file1 preceded with "< "
---
all 100 lines from file2 preceded with "> "
```

The ‘‘Only in’’ messages required when the **–r** option is specified are not used by most historical implementations if the **–e** option is also specified. It is required here because it provides useful information that must be provided to update a target directory hierarchy to match a source hierarchy. The ‘‘Common subdirectories’’ messages are written by System V and 4.3 BSD when the **–r** option is specified. They are allowed here but are not required because they are reporting on something that is the same, not reporting a difference, and are not needed to update a target hierarchy.

The **–c** option, which writes output in a format using lines of context, has been included. The format is useful for a variety of reasons, among them being much improved readability and the ability to understand difference changes when the target file has line numbers that differ from another similar, but slightly different, copy. The *patch* utility is most valuable when working with difference listings using the context format. The BSD version of **–c** takes an optional argument specifying the amount of context. Rather than overloading **–c** and breaking the Utility Syntax Guidelines for *diff*, the standard developers decided to add a separate option for specifying a context diff with a specified amount of context (**–C**). Also, the format for context *diff*s was extended slightly in 4.3 BSD to allow multiple changes that are within context lines from each other to be merged together. The output format contains an additional four asterisks after the range of affected lines in the first filename. This was to provide a flag for old programs (like old versions of *patch*) that only understand the old context format. The version of context described here does not require that multiple changes within context lines be merged, but it does not prohibit it either. The extension is upward-compatible, so any vendors that wish to retain the old version of *diff* can do so by adding the extra four asterisks (that is, utilities that currently use *diff* and understand the new merged format will also understand the old unmerged format, but not *vice versa*).

The substitute command was added as an additional format for the **–e** option. This was added to provide implementations a way to fix the classic ‘‘dot alone on a line’’ bug present in many versions of *diff*. Since many implementations have fixed this bug, the standard developers decided not to standardize broken behavior, but rather to provide the necessary tool for fixing the bug. One way to fix this bug is to output two periods whenever a lone period is needed, then terminate the append command with a period, and then use the substitute command to convert the two periods into one period.

The BSD-derived **–r** option was added to provide a mechanism for using *diff* to compare two file system trees. This behavior is useful, is standard practice on all BSD-derived systems, and is not easily reproducible with the *find* utility.

The requirement that *diff* not compare files in some circumstances, even though they have the same name, is based on the actual output of historical implementations. The message specified here is already in use when a directory is being compared to a non-directory. It is extended here to preclude the problems arising from running into FIFOs and other files that would cause *diff* to hang waiting for input with no indication to the user that *diff* was hung. In most common usage, *diff* **−r** should indicate differences in the file hierarchies, not the difference of contents of devices pointed to by the hierarchies.

Many early implementations of *diff* require seekable files. Since the System Interfaces volume of IEEE Std 1003.1-200x supports named pipes, the standard developers decided that such a restriction was unreasonable. Note also that the allowed filename – almost always refers to a pipe.

No directory search order is specified for *diff*. The historical ordering is, in fact, not optimal, in that it prints out all of the differences at the current level, including the statements about all common subdirectories before recursing into those subdirectories.

The message:

```
"diff %s %s %s\n", <diff_options>, <filename1>, <filename2>
```

does not vary by locale because it is the representation of a command, not an English sentence.

FUTURE DIRECTIONS

None.

SEE ALSO

cmp, *comm*, *ed*

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **−f** option is added.

The output format for **−c** or **−C** format is changed to align with changes to the IEEE P1003.2b draft standard resulting from IEEE PASC Interpretation 1003.2 #71.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

dirname — return the directory portion of pathname

SYNOPSIS

```
dirname string
```

DESCRIPTION

The *string* operand shall be treated as a pathname, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.266, Pathname. The string *string* shall be converted to the name of the directory containing the filename corresponding to the last pathname component in *string*, performing actions equivalent to the following steps in order:

1. If *string* is //, skip steps 2 to 5.
2. If *string* consists entirely of slash characters, *string* shall be set to a single slash character. In this case, skip steps 3 to 8.
3. If there are any trailing slash characters in *string*, they shall be removed.
4. If there are no slash characters remaining in *string*, *string* shall be set to a single period character. In this case, skip steps 5 to 8.
5. If there are any trailing non-slash characters in *string*, they shall be removed.
6. If the remaining *string* is //, it is implementation-defined whether steps 7 and 8 are skipped or processed.
7. If there are any trailing slash characters in *string*, they shall be removed.
8. If the remaining *string* is empty, *string* shall be set to a single slash character.

The resulting string shall be written to standard output.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

string	A string.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *dirname*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *dirname* utility shall write a line to the standard output in the following format:

```
"%s\n", <resulting string>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The definition of *pathname* specifies implementation-defined behavior for pathnames starting with two slash characters. Therefore, applications shall not arbitrarily add slashes to the beginning of a pathname unless they can ensure that there are more or less than two or are prepared to deal with the implementation-defined consequences.

EXAMPLES

Command	Results
dirname /	/
dirname //	/ or //
dirname /a/b/	/*a*
dirname //a//b//	//*a*
dirname	Unspecified
dirname a	. ($? = 0)
dirname ""	. ($? = 0)
dirname /a	/
dirname /a/b	/*a*
dirname a/b	*a*

RATIONALE

The *dirname* utility originated in System III. It has evolved through the System V releases to a version that matches the requirements specified in this description in System V Release 3. 4.3 BSD and earlier versions did not include *dirname*.

The behaviors of *basename* and *dirname* in this volume of IEEE Std 1003.1-200x have been coordinated so that when *string* is a valid pathname:

```
$(basename "string")
```

would be a valid filename for the file in the directory:

```
$(dirname "string")
```

This would not work for the versions of these utilities in early proposals due to the way processing of trailing slashes was specified. Consideration was given to leaving processing unspecified if there were trailing slashes, but this cannot be done; the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.266, Pathname allows trailing slashes. The *basename* and *dirname* utilities have to specify consistent handling for all valid pathnames.

FUTURE DIRECTIONS

None.

SEE ALSO

basename, Section 2.5 (on page 2235)

CHANGE HISTORY

First released in Issue 2.

NAME

du — estimate file space usage

SYNOPSIS

UP

```
du [−a | −s][−kx][−H | −L][file ...]
```

DESCRIPTION

By default, the *du* utility shall write to standard output the size of the file space allocated to, and the size of the file space allocated to each subdirectory of, the file hierarchy rooted in each of the specified files. By default, when a symbolic link is encountered on the command line or in the file hierarchy, *du* shall count the size of the symbolic link (rather than the file referenced by the link), and shall not follow the link to another portion of the file hierarchy. The size of the file space allocated to a file of type directory shall be defined as the sum total of space allocated to all files in the file hierarchy rooted in the directory plus the space allocated to the directory itself.

When *du* cannot *stat*( ) files or *stat*( ) or read directories, it shall report an error condition and the final exit status is affected. Files with multiple links shall be counted and written for only one entry. The directory entry that is selected in the report is unspecified. By default, file sizes shall be written in 512-byte units, rounded up to the next 512-byte unit.

OPTIONS

The *du* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	In addition to the default output, report the size of each file not of type directory in the file hierarchy rooted in the specified file. Regardless of the presence of the **−a** option, non-directories given as *file* operands shall always be listed.
−H	If a symbolic link is specified on the command line, *du* shall count the size of the file or file hierarchy referenced by the link.
−k	Write the files sizes in units of 1 024 bytes, rather than the default 512-byte units.
−L	If a symbolic link is specified on the command line or encountered during the traversal of a file hierarchy, *du* shall count the size of the file or file hierarchy referenced by the link.
−s	Instead of the default output, report only the total sum for each of the specified files.
−x	When evaluating file sizes, evaluate only those files that have the same device as the file specified by the *file* operand.

Specifying more than one of the mutually-exclusive options **−H** and **−L** shall not be considered an error. The last option specified shall determine the behavior of the utility.

OPERANDS

The following operand shall be supported:

file	The pathname of a file whose size is to be written. If no *file* is specified, the current directory shall be used.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *du*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output from *du* shall consist of the amount of the space allocated to a file and the name of the file, in the following format:

```
"%d %s\n", <size>, <pathname>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The use of 512-byte units is historical practice and maintains compatibility with *ls* and other utilities in this volume of IEEE Std 1003.1-200x. This does not mandate that the file system itself be based on 512-byte blocks. The −**k** option was added as a compromise measure. It was agreed by the standard developers that 512 bytes was the best default unit because of its complete historical consistency on System V (*versus* the mixed 512 / 1 024-byte usage on BSD systems), and that a −**k** option to switch to 1 024-byte units was a good compromise. Users who prefer the 1 024-byte quantity can easily alias *du* to *du* −**k** without breaking the many historical scripts relying on the 512-byte units.

The −**b** option was added to an early proposal to provide a resolution to the situation where System V and BSD systems give figures for file sizes in *blocks*, which is an implementation-defined concept. (In common usage, the block size is 512 bytes for System V and 1 024 bytes for BSD systems.) However, −**b** was later deleted, since the default was eventually decided as 512-byte units.

Historical file systems provided no way to obtain exact figures for the space allocation given to files. There are two known areas of inaccuracies in historical file systems: cases of *indirect blocks* being used by the file system or *sparse* files yielding incorrectly high values. An indirect block is space used by the file system in the storage of the file, but that need not be counted in the space allocated to the file. A *sparse* file is one in which an *lseek*( ) call has been made to a position beyond the end of the file and data has subsequently been written at that point. A file system need not allocate all the intervening zero-filled blocks to such a file. It is up to the implementation to define exactly how accurate its methods are.

The −**a** and −**s** options were mutually-exclusive in the original version of *du*. The POSIX Shell and Utilities description is implied by the language in the SVID where −**s** is described as causing ‘‘only the grand total’’ to be reported. Some systems may produce output for −**sa**, but a Strictly Conforming POSIX Shell and Utilities Application cannot use that combination.

The −**a** and −**s** options were adopted from the SVID except that the System V behavior of not listing non-directories explicitly given as operands, unless the −**a** option is specified, was considered a bug; the BSD-based behavior (report for all operands) is mandated. The default behavior of *du* in the SVID with regard to reporting the failure to read files (it produces no messages) was considered counter-intuitive, and thus it was specified that the POSIX Shell and Utilities default behavior shall be to produce such messages. These messages can be turned off with shell redirection to achieve the System V behavior.

The −**x** option is historical practice on recent BSD systems. It has been adopted by this volume of IEEE Std 1003.1-200x because there was no other historical method of limiting the *du* search to a single file hierarchy. This limitation of the search is necessary to make it possible to obtain file space usage information about a file system on which other file systems are mounted, without having to resort to a lengthy *find* and *awk* script.

FUTURE DIRECTIONS

None.

SEE ALSO

ls

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

This utility is reinstated, as the LEGACY marking was incorrect in Issue 5.

The obsolescent −**r** option has been removed.

The Open Group Corrigendum U025/3 is applied. The *du* utility had incorrectly been marked LEGACY.

The −**H** and −**L** options for symbolic links are added as described in the IEEE P1003.2b draft standard.

NAME

echo — write arguments to standard output

SYNOPSIS

```
echo [string ...]
```

DESCRIPTION

The *echo* utility writes its arguments to standard output, followed by a <newline>. If there are no arguments, only the <newline> is written.

OPTIONS

The *echo* utility shall not recognize the "−−" argument in the manner specified by Guideline 10 of the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines; "−−" shall be recognized as a string operand.

Implementations shall not support any options.

OPERANDS

The following operands shall be supported:

string A string to be written to standard output. If any operand is −**n**, it shall be treated as a string, not an option. The following character sequences shall be recognized within any of the arguments:

`\a`	Write an <alert>.
`\b`	Write a <backspace>.
`\c`	Suppress the <newline> that otherwise follows the final argument in the output. All characters following the '\c' in the arguments shall be ignored.
`\f`	Write a <form-feed>.
`\n`	Write a <newline>.
`\r`	Write a <carriage-return>.
`\t`	Write a <tab>.
`\v`	Write a <vertical-tab>.
`\\`	Write a backslash character.
`\0`*num*	Write an 8-bit value that is the zero, one, two, or three-digit octal number *num*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *echo*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *echo* utility arguments shall be separated by single <space>s and a <newline> shall follow the last argument. Output transformations shall occur based on the escape sequences in the input. See the OPERANDS section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

In the ISO/IEC 9945-2:1993 standard, it was not possible to use *echo* portably across all systems that were not XSI-conformant unless both **−n** (as the first argument) and escape sequences were omitted.

The *printf* utility can be used portably to emulate any of the traditional behaviors of the *echo* utility as follows:

- The historic System V *echo* and the current requirements in this volume of IEEE Std 1003.1-200x are equivalent to:

  ```
  printf "%b\n" "$*"
  ```

- The BSD *echo* is equivalent to:

  ```
  if [ "X$1" = "X−n" ]
  then
      shift
      printf "%s" "$*"
  else
      printf "%s\n" "$*"
  fi
  ```

New applications are encouraged to use *printf* instead of *echo*.

EXAMPLES

None.

RATIONALE

The *echo* utility has not been made obsolescent because of its extremely widespread use in historical applications. Conforming applications that wish to do prompting without <newline>s or that could possibly be expecting to echo a −**n**, should use the *printf* utility derived from the Ninth Edition system.

As specified, *echo* writes its arguments in the simplest of ways. The two different historical versions of *echo* vary in fatally incompatible ways.

The BSD *echo* checks the first argument for the string −**n** which causes it to suppress the <newline> that would otherwise follow the final argument in the output.

The System V *echo* does not support any options, but allows escape sequences within its operands, as described in the OPERANDS section.

The *echo* utility does not support Utility Syntax Guideline 10 because historical applications depend on *echo* to echo *all* of its arguments, except for the −**n** option in the BSD version.

FUTURE DIRECTIONS

None.

SEE ALSO

printf

CHANGE HISTORY

First released in Issue 2.

Issue 5

In the OPTIONS section, the last sentence is changed to indicate that implementations ‘‘do not’’ support any options; in the previous issue this said ‘‘need not’’.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- A set of character sequences is defined as *string* operands.
- *LC_CTYPE* is added to the list of environment variables affecting *echo*.
- In the OPTIONS section, implementations shall not support any options.

NAME

ed — edit text

SYNOPSIS

```
ed [−p string][−s][file]
```

DESCRIPTION

The *ed* utility is a line-oriented text editor that uses two modes: *command mode* and *input mode*. In command mode the input characters shall be interpreted as commands, and in input mode they shall be interpreted as text. See the EXTENDED DESCRIPTION section.

OPTIONS

The *ed* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−p *string*	Use *string* as the prompt string when in command mode. By default, there shall be no prompt string.
−s	Suppress the writing of byte counts by **e**, **E**, **r**, and **w** commands and of the `'!'` prompt after a !*command*.

OPERANDS

The following operand shall be supported:

file	If the *file* argument is given, *ed* shall simulate an **e** command on the file named by the pathname, *file*, before accepting commands from the standard input. If the *file* operand is `'−'`, the results are unspecified.

STDIN

The standard input shall be a text file consisting of commands, as described in the EXTENDED DESCRIPTION section.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ed*:

HOME	Determine the pathname of the user's home directory.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes within regular expressions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of

diagnostic messages written to standard error and informative messages written to standard output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

The *ed* utility shall take the standard action for all signals (see the ASYNCHRONOUS EVENTS section in Section 1.11 (on page 2221)) with the following exceptions:

SIGINT	The *ed* utility shall interrupt its current activity, write the string `"?\n"` to standard output, and return to command mode (see the EXTENDED DESCRIPTION section).
SIGHUP	If the buffer is not empty and has changed since the last write, the *ed* utility shall attempt to write a copy of the buffer in a file. First, the file named **ed.hup** in the current directory shall be used; if that fails, the file named **ed.hup** in the directory named by the *HOME* environment variable shall be used. In any case, the *ed* utility shall exit without returning to command mode.
SIGQUIT	The *ed* utility shall ignore this event.

STDOUT

Various editing commands and the prompting feature (see −**p**) write to standard output, as described in the EXTENDED DESCRIPTION section.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

The output files shall be text files whose formats are dependent on the editing commands given.

EXTENDED DESCRIPTION

The *ed* utility shall operate on a copy of the file it is editing; changes made to the copy shall have no effect on the file until a **w** (write) command is given. The copy of the text is called the *buffer*.

Commands to *ed* have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses very often can be omitted. If the −**p** option is specified, the prompt string shall be written to standard output before each command is read.

In general, only one command can appear on a line. Certain commands allow text to be input. This text is placed in the appropriate place in the buffer. While *ed* is accepting text, it is said to be in *input mode*. In this mode, no commands shall be recognized; all input is merely collected. Input mode is terminated by entering a line consisting of two characters: a period (`'.'`) followed by a <newline>. This line is not considered part of the input text.

Regular Expressions in ed

The *ed* utility shall support basic regular expressions, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions. Since regular expressions in *ed* are always matched against single lines (excluding the terminating <newline>s), never against any larger section of text, there is no way for a regular expression to match a <newline>.

A null RE shall be equivalent to the last RE encountered.

Regular expressions are used in addresses to specify lines, and in some commands (for example, the **s** substitute command) to specify portions of a line to be substituted.

Addresses in ed

Addressing in *ed* relates to the current line. Generally, the current line is the last line affected by a command. The current line number is the address of the current line. If the edit buffer is not empty, the initial value for the current line shall be the last line in the edit buffer; otherwise, zero.

Addresses shall be constructed as follows:

1. The period character (`'.'`) shall address the current line.
2. The dollar sign character (`'$'`) shall address the last line of the edit buffer.
3. The positive decimal number *n* shall address the *n*th line of the edit buffer.
4. The apostrophe-x character pair (`"'x"`) shall address the line marked with the mark name character *x*, which shall be a lowercase letter from the portable character set. It shall be an error if the character has not been set to mark a line or if the line that was marked is not currently present in the edit buffer.
5. A BRE enclosed by slash characters (`'/'`) shall address the first line found by searching forwards from the line following the current line toward the end of the edit buffer and stopping at the first line for which the line excluding the terminating <newline> matches the BRE. The BRE consisting of a null BRE delimited by a pair of slash characters shall address the next line for which the line excluding the terminating <newline> matches the last BRE encountered. In addition, the second slash can be omitted at the end of a command line. Within the BRE, a backslash-slash pair (`"\/"`) shall represent a literal slash instead of the BRE delimiter. If necessary, the search shall wrap around to the beginning of the buffer and continue up to and including the current line, so that the entire buffer is searched.
6. A BRE enclosed by question-mark characters (`'?'`) shall address the first line found by searching backwards from the line preceding the current line toward the beginning of the edit buffer and stopping at the first line for which the line excluding the terminating <newline> matches the BRE. The BRE consisting of a null BRE delimited by a pair of question-mark characters (`"??"`) shall address the previous line for which the line excluding the terminating <newline> matches the last BRE encountered. In addition, the second question-mark can be omitted at the end of a command line. Within the BRE, a backslash-question-mark pair (`"\?"`) shall represent a literal question mark instead of the BRE delimiter. If necessary, the search shall wrap around to the end of the buffer and continue up to and including the current line, so that the entire buffer is searched.
7. A plus-sign (`'+'`) or hyphen character (`'−'`) followed by a decimal number shall address the current line plus or minus the number. A plus-sign or hyphen character not followed by a decimal number shall address the current line plus or minus 1.

Addresses can be followed by zero or more address offsets, optionally <blank>-separated. Address offsets are constructed as follows:

- A plus-sign or hyphen character followed by a decimal number shall add or subtract, respectively, the indicated number of lines to or from the address. A plus-sign or hyphen character not followed by a decimal number shall add or subtract 1 to or from the address.
- A decimal number shall add the indicated number of lines to the address.

It shall not be an error for an intermediate address value to be less than zero or greater than the last line in the edit buffer. It shall be an error for the final address value to be less than zero or greater than the last line in the edit buffer. It shall be an error if a search for a BRE fails to find a matching line.

Commands accept zero, one, or two addresses. If more than the required number of addresses are provided to a command that requires zero addresses, it shall be an error. Otherwise, if more than the required number of addresses are provided to a command, the addresses specified first shall be evaluated and then discarded until the maximum number of valid addresses remain, for the specified command.

Addresses shall be separated from each other by a comma (`','`) or semicolon character (`';'`). In the case of a semicolon separator, the current line (`'.'`) shall be set to the first address, and only then will the second address be calculated. This feature can be used to determine the starting line for forwards and backwards searches; see rules 5. and 6.

Addresses can be omitted on either side of the comma or semicolon separator, in which case the resulting address pairs shall be as follows:

Specified	Resulting
`,`	`1 , $`
`, addr`	`1 ,a ddr`
`addr ,`	`addr , addr`
`;`	`. ; $`
`; addr`	`. ; addr`
`addr ;`	`addr ; addr`

Any <blank>s included between addresses, address separators, or address offsets shall be ignored.

Commands in ed

In the following list of *ed* commands, the default addresses are shown in parentheses. The number of addresses shown in the default shall be the number expected by the command. The parentheses are not part of the address; they show that the given addresses are the default.

It is generally invalid for more than one command to appear on a line. However, any command (except **e**, **E**, **f**, **q**, **Q**, **r**, **w**, and **!**) can be suffixed by the letter **l**, **n**, or **p**; in which case, except for the **l**, **n**, and **p** commands, the command shall be executed and then the new current line shall be written as described below under the **l**, **n**, and **p** commands. When an **l**, **n**, or **p** suffix is used with an **l**, **n**, or **p** command, the command shall write to standard output as described below, but it is unspecified whether the suffix writes the current line again in the requested format or whether the suffix has no effect. For example, the **pl** command (base **p** command with an **l** suffix) shall either write just the current line or write it twice—once as specified for **p** and once as specified for **l**. Also, the **g**, **G**, **v**, and **V** commands shall take a command as a parameter.

Each address component can be preceded by zero or more <blank>s. The command letter can be preceded by zero or more <blank>s. If a suffix letter (**l**, **n**, or **p**) is given, the application shall ensure that it immediately follows the command.

The **e**, **E**, **f**, **r**, and **w** commands shall take an optional *file* parameter, separated from the command letter by one or more <blank>s.

If changes have been made in the buffer since the last **w** command that wrote the entire buffer, *ed* shall warn the user if an attempt is made to destroy the editor buffer via the **e** or **q** commands. The *ed* utility shall write the string:

```
"?\n"
```

(followed by an explanatory message if *help mode* has been enabled via the **H** command) to standard output and shall continue in command mode with the current line number unchanged. If the **e** or **q** command is repeated with no intervening command, it shall take effect.

If a terminal disconnect is detected:

- If the buffer is not empty and has changed since the last write, the *ed* utility shall attempt to write a copy of the buffer to a file named **ed.hup** in the current directory. If this write fails, *ed* shall attempt to write a copy of the buffer to a filename **ed.hup** in the directory named by the *HOME* environment variable. If both these attempts fail, *ed* shall exit without saving the buffer.
- The *ed* utility shall not write the file to the currently remembered pathname or return to command mode, and shall terminate with a non-zero exit status.

If an end-of-file is detected on standard input:

- If the *ed* utility is in input mode, *ed* shall terminate input mode and return to command mode. It is unspecified if any partially entered lines (that is, input text without a terminating <newline>) are discarded from the input text.
- If the *ed* utility is in command mode, it shall act as if a **q** command had been entered.

If the closing delimiter of an RE or of a replacement string (for example, `'/'`) in a **g**, **G**, **s**, **v**, or **V** command would be the last character before a <newline>, that delimiter can be omitted, in which case the addressed line shall be written. For example, the following pairs of commands are equivalent:

```
s/s1/s2   s/s1/s2/p
g/s1      g/s1/p
?s1       ?s1?
```

If an invalid command is entered, *ed* shall write the string:

```
"?\n"
```

(followed by an explanatory message if *help mode* has been enabled via the **H** command) to standard output and shall continue in command mode with the current line number unchanged.

Append Command

Synopsis:

```
(.)a
<text>
.
```

The **a** command shall read the given text and append it after the addressed line; the current line number shall become the address of the last inserted line or, if there were none, the addressed line. Address 0 shall be valid for this command; it shall cause the appended text to be placed at the beginning of the buffer.

Change Command

Synopsis:

```
(.,.)c
<text>
.
```

The **c** command shall delete the addressed lines, then accept input text that replaces these lines; the current line shall be set to the address of the last line input; or, if there were none, at the line after the last line deleted; if the lines deleted were originally at the end of the buffer, the current line number shall be set to the address of the new last line; if no lines remain in the buffer, the current line number shall be set to zero. Address 0 shall be valid for this command; it shall be interpreted as if address 1 were specified.

Delete Command

Synopsis: `(.,.)d`

The **d** command shall delete the addressed lines from the buffer. The address of the line after the last line deleted shall become the current line number; if the lines deleted were originally at the end of the buffer, the current line number shall be set to the address of the new last line; if no lines remain in the buffer, the current line number shall be set to zero.

Edit Command

Synopsis: `e [file]`

The **e** command shall delete the entire contents of the buffer and then read in the file named by the pathname *file*. The current line number shall be set to the address of the last line of the buffer. If no pathname is given, the currently remembered pathname, if any, shall be used (see the **f** command). The number of bytes read shall be written to standard output, unless the **−s** option was specified, in the following format:

`"%d\n", <number of bytes read>`

The name *file* shall be remembered for possible use as a default pathname in subsequent **e**, **E**, **r**, and **w** commands. If *file* is replaced by `'!'`, the rest of the line shall be taken to be a shell command line whose output is to be read. Such a shell command line shall not be remembered as the current *file*. All marks shall be discarded upon the completion of a successful **e** command. If the buffer has changed since the last time the entire buffer was written, the user shall be warned, as described previously.

Edit Without Checking Command

Synopsis: `E [file]`

The **E** command shall possess all properties and restrictions of the **e** command except that the editor shall not check to see whether any changes have been made to the buffer since the last **w** command.

Filename Command

Synopsis: `f [file]`

If *file* is given, the **f** command shall change the currently remembered pathname to *file*; whether the name is changed or not, it shall then write the (possibly new) currently remembered pathname to the standard output in the following format:

`"%s\n", <pathname>`

The current line number shall be unchanged.

Global Command

Synopsis: `(1,$)g/RE/command list`

In the **g** command, the first step shall be to mark every line for which the line excluding the terminating <newline> matches the given *RE*. Then, going sequentially from the beginning of the file to the end of the file, the given *command list* shall be executed for each marked line, with the current line number set to the address of that line. Any line modified by the *command list* shall be unmarked. When the **g** command completes, the current line number shall have the value assigned by the last command in the *command list*. If there were no matching lines, the current line number shall not be changed. A single command or the first of a list of commands shall appear on the same line as the global command. All lines of a multi-line list except the last line shall be ended with a backslash preceding the terminating <newline>; the **a**, **i**, and **c** commands and associated input are permitted. The `'.'` terminating input mode can be omitted if it would be the last line of the *command list*. An empty *command list* shall be equivalent to the **p** command. The use of the **g**, **G**, **v**, **V**, and **!** commands in the *command list* produces undefined results. Any character other than <space> or <newline> can be used instead of a slash to delimit the *RE*. Within the *RE*, the *RE* delimiter itself can be used as a literal character if it is preceded by a backslash.

Interactive Global Command

Synopsis: `(1,$)G/`*`RE`*`/`

In the **G** command, the first step shall be to mark every line for which the line excluding the terminating <newline> matches the given *RE*. Then, for every such line, that line shall be written, the current line number shall be set to the address of that line, and any one command (other than one of the **a**, **c**, **i**, **g**, **G**, **v**, and **V** commands) shall be read and executed. A <newline> shall act as a null command (causing no action to be taken on the current line); an `'&'` shall cause the re-execution of the most recent non-null command executed within the current invocation of **G**. Note that the commands input as part of the execution of the **G** command can address and affect any lines in the buffer. The final value of the current line number shall be the value set by the last command successfully executed. (Note that the last command successfully executed shall be the **G** command itself if a command fails or the null command is specified.) If there were no matching lines, the current line number shall not be changed. The **G** command can be terminated by a SIGINT signal. Any character other than <space> or <newline> can be used instead of a slash to delimit the *RE* and the replacement. Within the *RE*, the *RE* delimiter itself can be used as a literal character if it is preceded by a backslash.

Help Command

Synopsis: `h`

The **h** command shall write a short message to standard output that explains the reason for the most recent `'?'` notification. The current line number shall be unchanged.

Help-Mode Command

Synopsis: `H`

The **H** command shall cause *ed* to enter a mode in which help messages (see the **h** command) shall be written to standard output for all subsequent `'?'` notifications. The **H** command alternately shall turn this mode on and off; it is initially off. If the help-mode is being turned on, the **H** command also explains the previous `'?'` notification, if there was one. The current line number shall be unchanged.

Insert Command

Synopsis:

```
(.)i
<text>
.
```

The **i** command shall insert the given text before the addressed line; the current line is set to the last inserted line or, if there was none, to the addressed line. This command differs from the **a** command only in the placement of the input text. Address 0 shall be valid for this command; it shall be interpreted as if address 1 were specified.

Join Command

Synopsis: `(.,.+1)j`

The **j** command shall join contiguous lines by removing the appropriate <newline>s. If exactly one address is given, this command shall do nothing. If lines are joined, the current line number shall be set to the address of the joined line; otherwise, the current line number shall be unchanged.

Mark Command

Synopsis: `(.)k`*`x`*

The **k** command shall mark the addressed line with name *x*, which the application shall ensure is a lowercase letter from the portable character set. The address `"'x"` shall then refer to this line; the current line number shall be unchanged.

List Command

Synopsis: `(.,.)l`

The **l** command shall write to standard output the addressed lines in a visually unambiguous form. The characters listed in the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\r'`, `'\t'`, `'\v'`) shall be written as the corresponding escape sequence; the `'\n'` in that table is not applicable. Non-printable characters not in the table shall be written as one three-digit octal number (with a preceding backslash character) for each byte in the character (most significant byte first). If the size of a byte on the system is greater than nine bits, the format used for non-printable characters is implementation-defined.

Long lines shall be folded, with the point of folding indicated by <newline> preceded by a backslash; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line shall be marked with a `'$'`, and `'$'` characters within the text shall be written with a preceding backslash. An **l** command can be appended to any other command other than **e**, **E**, **f**, **q**, **Q**, **r**, **w**, or **!**. The current line number shall be set to the address of the last line written.

Move Command

Synopsis: `(.,.)m`*`address`*

The **m** command shall reposition the addressed lines after the line addressed by *address*. Address 0 shall be valid for *address* and cause the addressed lines to be moved to the beginning of the buffer. It shall be an error if address *address* falls within the range of moved lines. The current line number shall be set to the address of the last line moved.

Number Command

Synopsis: `(.,.)n`

The **n** command shall write to standard output the addressed lines, preceding each line by its line number and a <tab>; the current line number shall be set to the address of the last line written. The **n** command can be appended to any command other than **e**, **E**, **f**, **q**, **Q**, **r**, **w**, or **!**.

Print Command

Synopsis: `(.,.)p`

The **p** command shall write to standard output the addressed lines; the current line number shall be set to the address of the last line written. The **p** command can be appended to any command other than **e**, **E**, **f**, **q**, **Q**, **r**, **w**, or **!**.

Prompt Command

Synopsis: `P`

The **P** command shall cause *ed* to prompt with an asterisk (`'*'`) (or *string*, if −**p** is specified) for all subsequent commands. The **P** command alternatively shall turn this mode on and off; it shall be initially on if the −**p** option is specified; otherwise, off. The current line number shall be unchanged.

Quit Command

Synopsis: `q`

The **q** command shall cause *ed* to exit. If the buffer has changed since the last time the entire buffer was written, the user shall be warned, as described previously.

Quit Without Checking Command

Synopsis: `Q`

The **Q** command shall cause *ed* to exit without checking whether changes have been made in the buffer since the last **w** command.

Read Command

Synopsis: `($)r` **[***file***]**

The **r** command shall read in the file named by the pathname *file* and append it after the addressed line. If no *file* argument is given, the currently remembered pathname, if any, shall be used (see the **e** and **f** commands). The currently remembered pathname shall not be changed unless there is no remembered pathname. Address 0 shall be valid for **r** and shall cause the file to be read at the beginning of the buffer. If the read is successful, and −**s** was not specified, the number of bytes read shall be written to standard output in the following format:

```
"%d\n", <number of bytes read>
```

The current line number shall be set to the address of the last line read in. If *file* is replaced by `'!'`, the rest of the line shall be taken to be a shell command line whose output is to be read. Such a shell command line shall not be remembered as the current pathname.

Substitute Command

Synopsis: `(.,.)s/`*RE*`/`*replacement*`/`*flags*

The **s** command shall search each addressed line for an occurrence of the specified RE and replace either the first or all (non-overlapped) matched strings with the *replacement*; see the following description of the **g** suffix. It is an error if the substitution fails on every addressed line. Any character other than <space> or <newline> can be used instead of a slash to delimit the *RE* and the replacement. Within the *RE*, the *RE* delimiter itself can be used as a literal character if it is preceded by a backslash. The current line shall be set to the address of the last line on which a substitution occurred.

An ampersand (`'&'`) appearing in the *replacement* shall be replaced by the string matching the RE on the current line. The special meaning of `'&'` in this context can be suppressed by preceding it by backslash. As a more general feature, the characters `'\n'`, where *n* is a digit, shall be replaced by the text matched by the corresponding back-reference expression. When the character `'%'` is the only character in the *replacement*, the *replacement* used in the most recent substitute command shall be used as the *replacement* in the current substitute command; if there was no previous substitute command, the use of `'%'` in this manner shall be an error. The `'%'` shall lose its special meaning when it is in a replacement string of more than one character or is preceded by a backslash. For each backslash (`'\'`) encountered in scanning *replacement* from beginning to end, the following character shall lose its special meaning (if any). It is unspecified what special meaning is given to any character other than `'&'`, `'\'`, `'%'`, or digits.

A line can be split by substituting a <newline> into it. The application shall ensure it escapes the <newline> in the *replacement* by preceding it by backslash. Such substitution cannot be done as part of a **g** or **v** *command list*. The current line number shall be set to the address of the last line on which a substitution is performed. If no substitution is performed, the current line number shall be unchanged. If a line is split, a substitution shall be considered to have been performed on each of the new lines for the purpose of determining the new current line number. A substitution shall be considered to have been performed even if the replacement string is identical to the string that it replaces.

The application shall ensure that the value of *flags* is zero or more of:

count	Substitute for the *count*th occurrence only of the *RE* found on each addressed line.
g	Globally substitute for all non-overlapping instances of the *RE* rather than just the first one. If both **g** and *count* are specified, the results are unspecified.
l	Write to standard output the final line in which a substitution was made. The line shall be written in the format specified for the **l** command.
n	Write to standard output the final line in which a substitution was made. The line shall be written in the format specified for the **n** command.
p	Write to standard output the final line in which a substitution was made. The line shall be written in the format specified for the **p** command.

Copy Command

Synopsis: `(.,.)taddress`

The **t** command shall be equivalent to the **m** command, except that a copy of the addressed lines shall be placed after address *address* (which can be 0); the current line number shall be set to the address of the last line added.

Undo Command

Synopsis: `u`

The **u** command shall nullify the effect of the most recent command that modified anything in the buffer, namely the most recent **a**, **c**, **d**, **g**, **i**, **j**, **m**, **r**, **s**, **t**, **u**, **v**, **G**, or **V** command. All changes made to the buffer by a **g**, **G**, **v**, or **V** global command shall be undone as a single change; if no changes were made by the global command (such as with **g**/*RE*/**p**), the **u** command shall have no effect. The current line number shall be set to the value it had immediately before the command being undone started.

Global Non-Matched Command

Synopsis: `(1,$)v/`*`RE`*`/`*`command list`*

This command shall be equivalent to the global command **g** except that the lines that are marked during the first step shall be those for which the line excluding the terminating <newline> does not match the *RE*.

Interactive Global Not-Matched Command

Synopsis: `(1,$)V/`*`RE`*`/`

This command shall be equivalent to the interactive global command **G** except that the lines that are marked during the first step shall be those for which the line excluding the terminating <newline> does not match the *RE*.

Write Command

Synopsis: `(1,$)w [`*`file`*`]`

The **w** command shall write the addressed lines into the file named by the pathname *file*. The command shall create the file, if it does not exist, or shall replace the contents of the existing file. The currently remembered pathname shall not be changed unless there is no remembered pathname. If no pathname is given, the currently remembered pathname, if any, shall be used (see the **e** and **f** commands); the current line number shall be unchanged. If the command is successful, the number of bytes written shall be written to standard output, unless the −**s** option was specified, in the following format:

`"%d\n", <`*`number of bytes written`*`>`

If *file* begins with `'!'`, the rest of the line shall be taken to be a shell command line whose standard input shall be the addressed lines. Such a shell command line shall not be remembered as the current pathname. This usage of the write command with `'!'` shall not be considered as a ‘‘last **w** command that wrote the entire buffer’’, as described previously; thus, this alone shall not prevent the warning to the user if an attempt is made to destroy the editor buffer via the **e** or **q** commands.

Line Number Command

Synopsis: `($)=`

The line number of the addressed line shall be written to standard output in the following format:

`"%d\n", <`*`line number`*`>`

The current line number shall be unchanged by this command.

Shell Escape Command

Synopsis: `!`*`command`*

The remainder of the line after the `'!'` shall be sent to the command interpreter to be interpreted as a shell command line. Within the text of that shell command line, the unescaped character `'%'` shall be replaced with the remembered pathname; if a `'!'` appears as the first character of the command, it shall be replaced with the text of the previous shell command executed via `'!'`. Thus, `"!!"` shall repeat the previous !*command*. If any replacements of `'%'` or `'!'` are performed, the modified line shall be written to the standard output before *command* is executed. The `'!'` command shall write:

`"!\n"`

to standard output upon completion, unless the **–s** option is specified. The current line number shall be unchanged.

Null Command

Synopsis: `(.+1)`

An address alone on a line shall cause the addressed line to be written. A <newline> alone shall be equivalent to `"+1p"`. The current line number shall be set to the address of the written line.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion without any file or command errors.
>0	An error occurred.

CONSEQUENCES OF ERRORS

When an error in the input script is encountered, or when an error is detected that is a consequence of the data (not) present in the file or due to an external condition such as a read or write error:

- If the standard input is a terminal device file, all input shall be flushed, and a new command read.
- If the standard input is a regular file, *ed* shall terminate with a non-zero exit status.

APPLICATION USAGE

Because of the extremely terse nature of the default error messages, the prudent script writer begins the *ed* input commands with an **H** command, so that if any errors do occur at least some clue as to the cause is made available.

In previous versions, an obsolescent – option was described. This is no longer specified. Applications should use the **–s** option. Using – as a *file* operand now produces unspecified results. This allows implementations to continue to support the former required behavior.

EXAMPLES

None.

RATIONALE

The initial description of this utility was adapted from the SVID. It contains some features not found in Version 7 or BSD-derived systems. Some of the differences between the POSIX and BSD *ed* utilities include, but need not be limited to:

- The BSD – option does not suppress the `'!'` prompt after a **!** command.
- BSD does not support the special meanings of the `'%'` and `'!'` characters within a **!** command.
- BSD does not support the *addresses* `';'` and `','`.
- BSD allows the command/suffix pairs **pp**, **ll**, and so on, which are unspecified in this volume of IEEE Std 1003.1-200x.
- BSD does not support the `'!'` character part of the **e**, **r**, or **w** commands.
- A failed **g** command in BSD sets the line number to the last line searched if there are no matches.

- BSD does not default the *command list* to the **p** command.
- BSD does not support the **G**, **h**, **H**, **n**, or **V** commands.
- On BSD, if there is no inserted text, the insert command changes the current line to the referenced line −1; that is, the line before the specified line.
- On BSD, the *join* command with only a single address changes the current line to that address.
- BSD does not support the **P** command; moreover, in BSD it is synonymous with the **p** command.
- BSD does not support the *undo* of the commands **j**, **m**, **r**, **s**, or **t**.
- The Version 7 *ed* command **W**, and the BSD *ed* commands **W**, **wq**, and **z** are not present in this volume of IEEE Std 1003.1-200x.

The −**s** option was added to allow the functionality of the now withdrawn – option in a manner compatible with the Utility Syntax Guidelines.

In early proposals there was a limit, {ED_FILE_MAX}, that described the historical limitations of some *ed* utilities in their handling of large files; some of these have had problems with files larger than 100 000 bytes. It was this limitation that prompted much of the desire to include a *split* command in this volume of IEEE Std 1003.1-200x. Since this limit was removed, this volume of IEEE Std 1003.1-200x requires that implementations document the file size limits imposed by *ed* in the conformance document. The limit {ED_LINE_MAX} was also removed; therefore, the global limit {LINE_MAX} is used for input and output lines.

The manner in which the **l** command writes non-printable characters was changed to avoid the historical backspace-overstrike method. On video display terminals, the overstrike is ambiguous because most terminals simply replace overstruck characters, making the **l** format not useful for its intended purpose of unambiguously understanding the content of the line. The historical backslash escapes were also ambiguous. (The string `"a\0011"` could represent a line containing those six characters or a line containing the three characters `'a'`, a byte with a binary value of 1, and a 1.) In the format required here, a backslash appearing in the line is written as `'\\'` so that the output is truly unambiguous. The method of marking the ends of lines was adopted from the *ex* editor and is required for any line ending in <space>s; the `'$'` is placed on all lines so that a real `'$'` at the end of a line cannot be misinterpreted.

Systems with bytes too large to fit into three octal digits must devise other means of displaying non-printable characters. Consideration was given to requiring that the number of octal digits be large enough to hold a byte, but this seemed to be too confusing for applications on the vast majority of systems where three digits are adequate. It would be theoretically possible for the application to use the *getconf* utility to find out the CHAR_BIT value and deal with such an algorithm; however, there is really no portable way that an application can use the octal values of the bytes across various coded character sets, so the additional specification was not worthwhile.

The description of how a NUL is written was removed. The NUL character cannot be in text files, and this volume of IEEE Std 1003.1-200x should not dictate behavior in the case of undefined, erroneous input.

Unlike some of the other editing utilities, the filenames accepted by the **E**, **e**, **R**, and **r** commands are not patterns.

Early proposals stated that the −**p** option worked only when standard input was associated with a terminal device. This has been changed to conform to historical implementations, thereby allowing applications to interpose themselves between a user and the *ed* utility.

The form of the substitute command that uses the **n** suffix was limited in some historical documentation (where this was described incorrectly as ‘‘backreferencing’’). This limit has been omitted because there is no reason an editor processing lines of {LINE_MAX} length should have this restriction. The command **s/x/X/2047** should be able to substitute the 2 047th occurrence of **x** on a line.

The use of printing commands with printing suffixes (such as **pn**, **lp**, and so on) was made unspecified because BSD-based systems allow this, whereas System V does not.

Some BSD-based systems exit immediately upon receipt of end-of-file if all of the lines in the file have been deleted. Since this volume of IEEE Std 1003.1-200x refers to the **q** command in this instance, such behavior is not allowed.

Some historical implementations returned exit status zero even if command errors had occurred; this is not allowed by this volume of IEEE Std 1003.1-200x.

Some historical implementations contained a bug that allowed a single period to be entered in input mode as <backslash> <period> <newline>. This is not allowed by the *ed* because there is no description of escaping any of the characters in input mode; backslashes are entered into the buffer exactly as typed. The typical method of entering a single period has been to precede it with another character and then use the substitute command to delete that character.

It is difficult under some modes of some versions of historical operating system terminal drivers to distinguish between an end-of-file condition and terminal disconnect. IEEE Std 1003.1-200x does not require implementations to distinguish between the two situations, which permits historical implementations of the *ed* utility on historical platforms to conform. Implementations are encouraged to distinguish between the two, if possible, and take appropriate action on terminal disconnect.

Historically, *ed* accepted a zero address for the **a** and **r** commands in order to insert text at the start of the edit buffer. When the buffer was empty the command **.=** returned zero. IEEE Std 1003.1-200x requires conformance to historical practice.

For consistency with the **a** and **r** commands and better user functionality, the **i** and **c** commands must also accept an address of 0, in which case 0*i* is treated as 1*i* and likewise for the **c** command.

All of the following are valid addresses:

`+++`	Three lines after the current line.
`/pattern/−`	One line before the next occurrence of pattern.
`−2`	Two lines before the current line.
`3 −−−− 2`	Line one (note the intermediate negative address).
`1 2 3`	Line six.

Any number of addresses can be provided to commands taking addresses; for example, `"1,2,3,4,5p"` prints lines 4 and 5, because two is the greatest valid number of addresses accepted by the **print** command. This, in combination with the semicolon delimiter, permits users to create commands based on ordered patterns in the file. For example, the command `"3;/foo/;+2p"` will display the first line after line 3 that contains the pattern *foo*, plus the next two lines. Note that the address `"3;"` must still be evaluated before being discarded, because the search origin for the `"/foo/"` command depends on this.

Historically, *ed* disallowed address chains, as discussed above, consisting solely of comma or semicolon separators; for example, `",,,"` or `";;;"` were considered an error. For consistency of address specification, this restriction is removed. The following table lists some of the address forms now possible:

Address	Addr1	Addr2	Status	Comment
`7,`	`7`	`7`	Historical	
`7,5,`	`5`	`5`	Historical	
`7,5,9`	`5`	`9`	Historical	
`7,9`	`7`	`9`	Historical	
`7,+`	`7`	`8`	Historical	
`,`	`1`	`$`	Historical	
`,7`	`1`	`7`	Extension	
`,,`	`$`	`$`	Extension	
`,;`	`$`	`$`	Extension	
`7;`	`7`	`7`	Historical	
`7;5;`	`5`	`5`	Historical	
`7;5;9`	`5`	`9`	Historical	
`7;5,9`	`5`	`9`	Historical	
`7;$;4`	`$`	`4`	Historical	Valid, but erroneous.
`7;9`	`7`	`9`	Historical	
`7;+`	`7`	`8`	Historical	
`;`	`.`	`$`	Historical	
`;7`	`.`	`7`	Extension	
`;;`	`$`	`$`	Extension	
`;,`	`$`	`$`	Extension	

Historically, values could be added to addresses by including them after one or more <blank>s; for example, `"3 – 5p"` wrote the seventh line of the file, and `"/foo/ 5"` was the same as `"5 /foo/"`. However, only absolute values could be added; for example, `"5 /foo/"` was an error. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, *ed* accepted the `'^'` character as an address, in which case it was identical to the hyphen character. IEEE Std 1003.1-200x does not require or prohibit this behavior.

FUTURE DIRECTIONS

None.

SEE ALSO

ex, *sed*, *sh*, *vi*

CHANGE HISTORY

First released in Issue 2.

Issue 5

In the OPTIONS section, the meaning of –**s** and – is clarified.

Second FUTURE DIRECTION added.

Issue 6

The obsolescent single-minus form has been removed.

A second APPLICATION USAGE note has been added.

The Open Group Corrigendum U025/2 is applied, correcting the description of the Edit section.

The *ed* utility is updated to align with the IEEE P1003.2b draft standard. This includes addition of the treatment of the SIGQUIT signal, changes to *ed* addressing, changes to processing when end-of-file is detected and when terminal disconnect is detected.

13426 The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

env — set the environment for command invocation

SYNOPSIS

```
env [−i][name=value]... [utility [argument...]]
```

DESCRIPTION

The *env* utility shall obtain the current environment, modify it according to its arguments, then invoke the utility named by the *utility* operand with the modified environment.

Optional arguments shall be passed to *utility*.

If no *utility* operand is specified, the resulting environment shall be written to the standard output, with one *name*=*value* pair per line.

OPTIONS

The *env* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−i	Invoke *utility* with exactly the environment specified by the arguments; the inherited environment shall be ignored completely.

OPERANDS

The following operands shall be supported:

name=*value*	Arguments of the form *name*=*value* shall modify the execution environment, and shall be placed into the inherited environment before the *utility* is invoked.
utility	The name of the utility to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 (on page 2266), the results are undefined.
argument	A string to pass as an argument for the invoked utility.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *env*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

PATH Determine the location of the *utility*, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables. If *PATH* is specified as a *name*=*value* operand to *env*, the *value* given shall be used in the search for *utility*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If no *utility* operand is specified, each *name*=*value* pair in the resulting environment shall be written in the form:

```
"%s=%s\n", <name>, <value>
```

If the *utility* operand is specified, the *env* utility shall not write to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If the *utility* utility is invoked, the exit status of *env* shall be the exit status of *utility*; otherwise, the *env* utility shall exit with one of the following values:

0	The *env* utility completed successfully.
1–125	An error occurred in the *env* utility.
126	The utility specified by *utility* was found but could not be invoked.
127	The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *command*, *env*, *nice*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

Historical implementations of the *env* utility use the *execvp*( ) or *execlp*( ) functions defined in the System Interfaces volume of IEEE Std 1003.1-200x to invoke the specified utility; this provides better performance and keeps users from having to escape characters with special meaning to the shell. Therefore, shell functions, special built-ins, and built-ins that are only provided by the shell are not found.

EXAMPLES

The following command:

```
env −i PATH=/mybin mygrep xyz myfile
```

invokes the command *mygrep* with a new *PATH* value as the only entry in its environment. In this case, *PATH* is used to locate *mygrep*, which then must reside in **/mybin**.

RATIONALE

As with all other utilities that invoke other utilities, this volume of IEEE Std 1003.1-200x only specifies what *env* does with standard input, standard output, standard error, input files, and output files. If a utility is executed, it is not constrained by the specification of input and output by *env*.

The **−i** option was added to allow the functionality of the withdrawn − option in a manner compatible with the Utility Syntax Guidelines.

Some have suggested that *env* is redundant since the same effect is achieved by:

```
name=value ... utility [ argument ... ]
```

The example is equivalent to *env* when an environment variable is being added to the environment of the command, but not when the environment is being set to the given value. The *env* utility also writes out the current environment if invoked without arguments. There is sufficient functionality beyond what the example provides to justify inclusion of *env*.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.5 (on page 2235)

CHANGE HISTORY

First released in Issue 2.

NAME

ex — text editor

SYNOPSIS

UP `ex [−rR][−l][−s | −v][−c command][−t tagstring][−w size][file ...]`

DESCRIPTION

The *ex* utility is a line-oriented text editor. There are two other modes of the editor—open and visual—in which screen-oriented editing is available. This is described more fully by the *ex* **open** and **visual** commands and in *vi*.

This section uses the term *edit buffer* to describe the current working text. No specific implementation is implied by this term. All editing changes are performed on the edit buffer, and no changes to it shall affect any file until an editor command writes the file.

Certain terminals do not have all the capabilities necessary to support the complete *ex* definition, such as the full-screen editing commands (*visual mode* or *open mode*). When these commands cannot be supported on such terminals, this condition shall not produce an error message such as ''not an editor command'' or report a syntax error. The implementation may either accept the commands and produce results on the screen that are the result of an unsuccessful attempt to meet the requirements of this volume of IEEE Std 1003.1-200x or report an error describing the terminal-related deficiency.

OPTIONS

The *ex* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c *command* Specify an initial command to be executed in the first edit buffer loaded from an existing file (see the EXTENDED DESCRIPTION section). Implementations may support more than a single **−c** option. In such implementations, the specified commands shall be executed in the order specified on the command line.

−r Recover the named files (see the EXTENDED DESCRIPTION section). Recovery information for a file shall be saved during an editor or system crash (for example, when the editor is terminated by a signal which the editor can catch), or after the use of an *ex* **preserve** command.

A *crash* in this context is an unexpected failure of the system or utility that requires restarting the failed system or utility. A system crash implies that any utilities running at the time also crash. In the case of an editor or system crash, the number of changes to the edit buffer (since the most recent **preserve** command) that will be recovered is unspecified.

If no *file* operands are given and the **−t** option is not specified, all other options, the *EXINIT* variable, and any **.exrc** files shall be ignored; a list of all recoverable files available to the invoking user shall be written, and the editor shall exit normally without further action.

−R Set **readonly** edit option.

−s Prepare *ex* for batch use by taking the following actions:

- Suppress writing prompts and informational (but not diagnostic) messages.
- Ignore the value of *TERM* and any implementation default terminal type and assume the terminal is a type incapable of supporting open or visual modes;

see the **visual** command and the description of *vi*.

- Suppress the use of the *EXINIT* environment variable and the reading of any **.exrc** file; see the EXTENDED DESCRIPTION section.
- Suppress autoindentation, ignoring the value of the **autoindent** edit option.

–t *tagstring*	Edit the file containing the specified *tagstring*; see *ctags*. The tags feature represented by **–t** *tagstring* and the **tag** command is optional. It shall be provided on any system that also provides a conforming implementation of *ctags*; otherwise, the use of **–t** produces undefined results. On any system, it shall be an error to specify more than a single **–t** option.
–v	Begin in visual mode (see *vi*).
–w *size*	Set the value of the *window* editor option to *size*.

OPERANDS

The following operand shall be supported:

file	A pathname of a file to be edited.

STDIN

The standard input consists of a series of commands and input text, as described in the EXTENDED DESCRIPTION section. The implementation may limit each line of standard input to a length of {LINE_MAX}.

If the standard input is not a terminal device, it shall be as if the **–s** option had been specified.

If a read from the standard input returns an error, or if the editor detects an end-of-file condition from the standard input, it shall be equivalent to a SIGHUP asynchronous event.

INPUT FILES

Input files shall be text files or files that would be text files except for an incomplete last line that is not longer than {LINE_MAX}–1 bytes in length and contains no NUL characters. By default, any incomplete last line shall be treated as if it had a trailing <newline>. The editing of other forms of files may optionally be allowed by *ex* implementations.

The **.exrc** files and source files shall be text files consisting of *ex* commands; see the EXTENDED DESCRIPTION section.

By default, the editor shall read lines from the files to be edited without interpreting any of those lines as any form of editor command.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ex*:

COLUMNS	Override the system-selected horizontal screen size. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables for valid values and results when it is unset or null.
EXINIT	Determine a list of *ex* commands that are executed on editor start-up. See the EXTENDED DESCRIPTION section for more details of the initialization phase.
HOME	Determine a pathname of a directory that shall be searched for an editor start-up file named **.exrc**; see the EXTENDED DESCRIPTION section.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the behavior of character classes within regular expressions, the classification of characters as uppercase or lowercase letters, the case conversion of letters, and the detection of word boundaries.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LINES	Override the system-selected vertical screen size, used as the number of lines in a screenful and the vertical screen size in visual mode. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables for valid values and results when it is unset or null.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the search path for the shell command specified in the *ex* editor commands **!**, **shell**, **read**, and **write**, and the open and visual mode command **!**; see the description of command search and execution in Section 2.9.1.1 (on page 2249).
SHELL	Determine the preferred command line interpreter for use as the default value of the **shell** edit option.
TERM	Determine the name of the terminal type. If this variable is unset or null, an unspecified default terminal type shall be used.

ASYNCHRONOUS EVENTS

The following term is used in this and following sections to specify command and asynchronous event actions:

complete write

A complete write is a write of the entire contents of the edit buffer to a file of a type other than a terminal device, or the saving of the edit buffer caused by the user executing the *ex* **preserve** command. Writing the contents of the edit buffer to a temporary file that will be removed when the editor exits shall not be considered a complete write.

The following actions shall be taken upon receipt of signals:

SIGINT

If the standard input is not a terminal device, *ex* shall not write the file or return to command or text input mode, and shall exit with a non-zero exit status.

Otherwise, if executing an open or visual text input mode command, *ex* in receipt of SIGINT shall behave identically to its receipt of the <ESC> character.

Otherwise:

1. If executing an *ex* text input mode command, all input lines that have been completely entered shall be resolved into the edit buffer, and any partially entered line shall be discarded.

2. If there is a currently executing command, it shall be aborted and a message displayed. Unless otherwise specified by the *ex* or *vi* command descriptions, it is unspecified whether any lines modified by the executing command appear modified, or as they were before being modified by the executing command, in the buffer.

   If the currently executing command was a motion command, its associated command shall be discarded.

3. If in open or visual command mode, the terminal shall be alerted.

4. The editor shall then return to command mode.

SIGCONT	The screen shall be refreshed if in open or visual mode.
SIGHUP	If the edit buffer has been modified since the last complete write, *ex* shall attempt to save the edit buffer so that it can be recovered later using the **–r** option or the *ex* **recover** command. The editor shall not write the file or return to command or text input mode, and shall terminate with a non-zero exit status.
SIGTERM	Refer to SIGHUP.

The action taken for all other signals is unspecified.

STDOUT

The standard output shall be used only for writing prompts to the user, for informational messages, and for writing lines from the file.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output from *ex* shall be text files.

EXTENDED DESCRIPTION

Only the *ex* mode of the editor is described in this section. See *vi* for additional editing capabilities available in *ex*.

When an error occurs, *ex* shall write a message. If the terminal supports a standout mode (such as inverse video), the message shall be written in standout mode. If the terminal does not support a standout mode, and the edit option **errorbells** is set, an alert action shall precede the error message.

By default, *ex* shall start in command mode, which shall be indicated by a **:** prompt; see the **prompt** command. Text input mode can be entered by the **append**, **insert**, or **change** commands; it can be exited (and command mode re-entered) by typing a period (**'.'**) alone at the beginning of a line.

Initialization in ex and vi

The following symbols are used in this and following sections to specify locations in the edit buffer:

alternate and current path names

Two pathnames, named *current* and *alternate*, are maintained by the editor. Any *ex* commands that take filenames as arguments shall set them as follows:

1. If a *file* argument is specified to the *ex* **edit**, **ex**, or **recover** commands, or if an *ex* **tag** command replaces the contents of the edit buffer.

   a. If the command replaces the contents of the edit buffer, the current pathname shall be set to the *file* argument or the file indicated by the tag, and the alternate pathname shall be set to the previous value of the current pathname.

   b. Otherwise, the alternate pathname shall be set to the *file* argument.

2. If a *file* argument is specified to the *ex* **next** command:

   a. If the command replaces the contents of the edit buffer, the current pathname shall be set to the first *file* argument, and the alternate pathname shall be set to the previous value of the current pathname.

3. If a *file* argument is specified to the *ex* **file** command, the current pathname shall be set to the *file* argument, and the alternate pathname shall be set to the previous value of the current pathname.

4. If a *file* argument is specified to the *ex* **read** and **write** commands (that is, when reading or writing a file, and not to the program named by the **shell** edit option), or a *file* argument is specified to the *ex* **xit** command:

   a. If the current pathname has no value, the current pathname shall be set to the *file* argument.

   b. Otherwise, the alternate pathname shall be set to the *file* argument.

If the alternate pathname is set to the previous value of the current pathname when the current pathname had no previous value, then the alternate pathname shall have no value as a result.

current line
The line of the edit buffer referenced by the cursor. Each command description specifies the current line after the command has been executed, as the *current line value*. When the edit buffer contains no lines, the current line shall be zero; see **Addressing in ex** (on page 2562).

current column
The current display line column occupied by the cursor. (The columns shall be numbered beginning at 1.) Each command description specifies the current column after the command has been executed, as the *current column* value. This column is an *ideal* column that is remembered over the lifetime of the editor. The actual display line column upon which the cursor rests may be different from the current column; see the cursor positioning discussion in **Command Descriptions in vi** (on page 3186).

set to non-<blank>
A description for a current column value, meaning that the current column shall be set to the last display line column on which is displayed any part of the first non-<blank> of the line. If the line has no non-<blank> non- <newline>s, the current column shall be set to the last display line column on which is displayed any part of the last non-<newline> in the line. If the line is empty, the current column shall be set to column position 1.

The length of lines in the edit buffer may be limited to {LINE_MAX} bytes. In open and visual mode, the length of lines in the edit buffer may be limited to the number of characters that will fit in the display. If either limit is exceeded during editing, an error message shall be written. If either limit is exceeded by a line read in from a file, an error message shall be written and the edit session may be terminated.

If the editor stops running due to any reason other than a user command, and the edit buffer has been modified since the last complete write, it shall be equivalent to a SIGHUP asynchronous event. If the system crashes, it shall be equivalent to a SIGHUP asynchronous event.

During initialization (before the first file is copied into the edit buffer or any user commands from the terminal are processed) the following shall occur:

1. If the environment variable *EXINIT* is set, the editor shall execute the *ex* commands contained in that variable.
2. If the *EXINIT* variable is not set, and all of the following are true:
   a. The *HOME* environment variable is not null and not empty.
   b. The file **.exrc** in the directory referred to by the *HOME* environment variable:
      1. Exists
      2. Is owned by the same user ID as the real user ID of the process or the process has appropriate privileges
      3. Is not writeable by anyone other than the owner

   the editor shall execute the *ex* commands contained in that file.
3. If and only if all the following are true:
   a. The current directory is not referred to by the *HOME* environment variable.
   b. A command in the *EXINIT* environment variable or a command in the **.exrc** file in the directory referred to by the *HOME* environment variable sets the editor option **exrc**.
   c. The **.exrc** file in the current directory:
      1. Exists
      2. Is owned by the same user ID as the real user ID of the process, or by one of a set of implementation-defined user IDs
      3. Is not writeable by anyone other than the owner

   the editor shall attempt to execute the *ex* commands contained in that file.

Lines in any **.exrc** file that are blank lines shall be ignored. If any **.exrc** file exists, but is not read for ownership or permission reasons, it shall be an error.

After the *EXINIT* variable and any **.exrc** files are processed, the first file specified by the user shall be edited, as follows:

1. If the user specified the **–t** option, the effect shall be as if the *ex* **tag** command was entered with the specified argument, with the exception that if tag processing does not result in a file to edit, the effect shall be as described in step 3. below.
2. Otherwise, if the user specified any command line *file* arguments, the effect shall be as if the *ex* **edit** command was entered with the first of those arguments as its *file* argument.
3. Otherwise, the effect shall be as if the *ex* **edit** command was entered with a nonexistent filename as its *file* argument. It is unspecified whether this action shall set the current pathname. In an implementation where this action does not set the current pathname, any editor command using the current pathname shall fail until an editor command sets the current pathname.

If the **–r** option was specified, the first time a file in the initial argument list or a file specified by the **–t** option is edited, if recovery information has previously been saved about it, that information shall be recovered and the editor shall behave as if the contents of the edit buffer have already been modified. If there are multiple instances of the file to be recovered, the one most recently saved shall be recovered, and an informational message that there are previous versions of the file that can be recovered shall be written. If no recovery information about a file is available, an informational message to this effect shall be written, and the edit shall proceed as usual.

If the −**c** option was specified, the first time a file that already exists (including a file that might not exist but for which recovery information is available, when the −**r** option is specified) replaces or initializes the contents of the edit buffer, the current line shall be set to the last line of the edit buffer, the current column shall be set to non-<blank>, and the *ex* commands specified with the −**c** option shall be executed. In this case, the current line and current column shall not be set as described for the command associated with the replacement or initialization of the edit buffer contents. However, if the −**t** option or a **tag** command is associated with this action, the −**c** option commands shall be executed and then the movement to the tag shall be performed.

The current argument list shall initially be set to the filenames specified by the user on the command line. If no filenames are specified by the user, the current argument list shall be empty. If the −**t** option was specified, it is unspecified whether any filename resulting from tag processing shall be prepended to the current argument list. In the case where the filename is added as a prefix to the current argument list, the current argument list reference shall be set to that filename. In the case where the filename is not added as a prefix to the current argument list, the current argument list reference shall logically be located before the first of the filenames specified on the command line (for example, a subsequent *ex* **next** command shall edit the first filename from the command line). If the −**t** option was not specified, the current argument list reference shall be to the first of the filenames on the command line.

Addressing in ex

Addressing in *ex* relates to the current line and the current column; the address of a line is its 1-based line number, the address of a column is its 1-based count from the beginning of the line. Generally, the current line is the last line affected by a command. The current line number is the address of the current line. In each command description, the effect of the command on the current line number and the current column is described.

Addresses are constructed as follows:

1. The character '.' (period) shall address the current line.
2. The character '$' shall address the last line of the edit buffer.
3. The positive decimal number *n* shall address the *n*th line of the edit buffer.
4. The address "'x" refers to the line marked with the mark name character 'x', which shall be a lowercase letter from the portable character set or one of the characters '`' or '''. It shall be an error if the line that was marked is not currently present in the edit buffer or the mark has not been set. Lines can be marked with the *ex* **mark** or **k** commands, or the *vi* **m** command.
5. A regular expression (RE) enclosed by slashes ('/') shall address the first line found by searching forwards from the line following the current line toward the end of the edit buffer and stopping at the first line for which the line excluding the terminating <newline> matches the regular expression. As stated in **Regular Expressions in ex** (on page 2592), an address consisting of a null regular expression delimited by slashes "//" shall address the next line for which the line excluding the terminating <newline> matches the last regular expression encountered. In addition, the second slash can be omitted at the end of a command line. If the **wrapscan** edit option is set, the search shall wrap around to the beginning of the edit buffer and continue up to and including the current line, so that the entire edit buffer is searched. Within the regular expression, the sequence "\/" shall represent a literal slash instead of the regular expression delimiter.

6. A regular expression enclosed in question marks (`'?'`) shall address the first line found by searching backwards from the line preceding the current line toward the beginning of the edit buffer and stopping at the first line for which the line excluding the terminating <newline> matches the regular expression. An address consisting of a null regular expression delimited by question marks `"??"` shall address the previous line for which the line excluding the terminating <newline> matches the last regular expression encountered. In addition, the second question mark can be omitted at the end of a command line. If the **wrapscan** edit option is set, the search shall wrap around from the beginning of the edit buffer to the end of the edit buffer and continue up to and including the current line, so that the entire edit buffer is searched. Within the regular expression, the sequence `"\?"` shall represent a literal question mark instead of the RE delimiter.

7. A plus sign (`'+'`) or a minus sign (`'−'`) followed by a decimal number shall address the current line plus or minus the number. A `'+'` or `'−'` not followed by a decimal number shall address the current line plus or minus 1.

Addresses can be followed by zero or more address offsets, optionally <blank>-separated. Address offsets are constructed as follows:

1. A `'+'` or `'−'` immediately followed by a decimal number shall add (subtract) the indicated number of lines to (from) the address. A `'+'` or `'−'` not followed by a decimal number shall add (subtract) 1 to (from) the address.

2. A decimal number shall add the indicated number of lines to the address.

It shall not be an error for an intermediate address value to be less than zero or greater than the last line in the edit buffer. It shall be an error for the final address value to be less than zero or greater than the last line in the edit buffer.

Commands take zero, one, or two addresses; see the descriptions of *1addr* and *2addr* in **Command Descriptions in ex** (on page 2569). If more than the required number of addresses are provided to a command that requires zero addresses, it shall be an error. Otherwise, if more than the required number of addresses are provided to a command, the addresses specified first shall be evaluated and then discarded until the maximum number of valid addresses remain.

Addresses shall be separated from each other by a comma (`','`) or a semicolon (`';'`). If no address is specified before or after a comma or semicolon separator, it shall be as if the address of the current line was specified before or after the separator. In the case of a semicolon separator, the current line (`'.'`) shall be set to the first address, and only then will the next address be calculated. This feature can be used to determine the starting line for forwards and backwards searches (see rules 5. and 6.).

A percent sign (`'%'`) shall be equivalent to entering the two addresses `"1,$"`.

Any delimiting <blank>s between addresses, address separators, or address offsets shall be discarded.

Command Line Parsing in ex

The following symbol is used in this and following sections to describe parsing behavior:

escape If a character is referred to as ‘‘backslash escaped’’ or ‘‘<control>-V escaped,’’ it shall mean that the character acquired or lost a special meaning by virtue of being preceded, respectively, by a backslash or <control>-V character. Unless otherwise specified, the escaping character shall be discarded at that time and shall not be further considered for any purpose.

Command-line parsing shall be done in the following steps. For each step, characters already evaluated shall be ignored; that is, the phrase ‘‘leading character’’ refers to the next character that has not yet been evaluated.

1. Leading colon characters shall be skipped.
2. Leading <blank>s shall be skipped.
3. If the leading character is a double-quote character, the characters up to and including the next non-backslash-escaped <newline> shall be discarded, and any subsequent characters shall be parsed as a separate command.
4. Leading characters that can be interpreted as addresses shall be evaluated; see **Addressing in ex** (on page 2562).
5. Leading <blank>s shall be skipped.
6. If the next character is a vertical-line character or a <newline>:
   a. If the next character is a <newline>:
      1. If *ex* is in open or visual mode, the current line shall be set to the last address specified, if any.
      2. Otherwise, if the last command was terminated by a vertical-line character, no action shall be taken; for example, the command `"||<newline>"` shall execute two implied commands, not three.
      3. Otherwise, step 6.b. shall apply.

   b. Otherwise, the implied command shall be the **print** command. The last **#**, **p**, and **l** flags specified to any *ex* command shall be remembered and shall apply to this implied command. Executing the *ex* **number**, **print**, or **list** command shall set the remembered flags to **#**, nothing, and **l**, respectively, plus any other flags specified for that execution of the **number**, **print**, or **list** command.

      If *ex* is not currently performing a **global** or **v** command, and no address or count is specified, the current line shall be incremented by 1 before the command is executed. If incrementing the current line would result in an address past the last line in the edit buffer, the command shall fail, and the increment shall not happen.

   c. The <newline> or vertical-line character shall be discarded and any subsequent characters shall be parsed as a separate command.
7. The command name shall be comprised of the next character (if the character is not alphabetic), or the next character and any subsequent alphabetic characters (if the character is alphabetic), with the following exceptions:
   a. Commands that consist of any prefix of the characters in the command name **delete**, followed immediately by any of the characters `'l'`, `'p'`, `'+'`, `'-'`, or `'#'` shall be interpreted as a **delete** command, followed by a <blank>, followed by the characters that were not part of the prefix of the **delete** command. The maximum number of characters shall be matched to the command name **delete**; for example, `"del"` shall not be treated as `"de"` followed by the flag **l**.

   b. Commands that consist of the character `'k'`, followed by a character that can be used as the name of a mark, shall be equivalent to the mark command followed by a <blank>, followed by the character that followed the `'k'`.

   c. Commands that consist of the character `'s'`, followed by characters that could be interpreted as valid options to the **s** command, shall be the equivalent of the **s** command, without any pattern or replacement values, followed by a <blank>, followed by the characters after the `'s'`.

8. The command name shall be matched against the possible command names, and a command name that contains a prefix matching the characters specified by the user shall be the executed command. In the case of commands where the characters specified by the user could be ambiguous, the executed command shall be as follows:

a	**append**	**n**	**next**	**t**	**t**
c	**change**	**p**	**print**	**u**	**undo**
ch	**change**	**pr**	**print**	**un**	**undo**
e	**edit**	**r**	**read**	**v**	**v**
m	**move**	**re**	**read**	**w**	**write**
ma	**mark**	**s**	**s**		

Implementation extensions with names causing similar ambiguities shall not be checked for a match until all possible matches for commands specified by IEEE Std 1003.1-200x have been checked.

9. If the command is a **!** command, or if the command is a **read** command followed by zero or more <blank>s and a **!**, or if the command is a **write** command followed by one or more <blank>s and a **!**, the rest of the command shall include all characters up to a non-backslash-escaped <newline>. The <newline> shall be discarded and any subsequent characters shall be parsed as a separate *ex* command.

10. Otherwise, if the command is an **edit**, **ex**, or **next** command, or a **visual** command while in open or visual mode, the next part of the command shall be parsed as follows:

    a. Any `'!'` character immediately following the command shall be skipped and be part of the command.

    b. Any leading <blank>s shall be skipped and be part of the command.

    c. If the next character is a `'+'`, characters up to the first non-backslash-escaped <newline> or non-backslash-escaped <blank> shall be skipped and be part of the command.

    d. The rest of the command shall be determined by the steps specified in paragraph 12.

11. Otherwise, if the command is a **global**, **open**, **s**, or **v** command, the next part of the command shall be parsed as follows:

    a. Any leading <blank>s shall be skipped and be part of the command.

    b. If the next character is not an alphanumeric, double-quote, <newline>, backslash, or vertical-line character:

        1. The next character shall be used as a command delimiter.

        2. If the command is a **global**, **open**, or **v** command, characters up to the first non-backslash-escaped <newline>, or first non-backslash-escaped delimiter character, shall be skipped and be part of the command.

        3. If the command is an **s** command, characters up to the first non-backslash-escaped <newline>, or second non-backslash-escaped delimiter character, shall be skipped and be part of the command.

    c. If the command is a **global** or **v** command, characters up to the first non-backslash-escaped <newline> shall be skipped and be part of the command.

  d. Otherwise, the rest of the command shall be determined by the steps specified in paragraph 12.

12. Otherwise:

  a. If the command was a **map**, **unmap**, **abbreviate**, or **unabbreviate** command, characters up to the first non-<control>-V-escaped <newline>, vertical-line, or double-quote character shall be skipped and be part of the command.

  b. Otherwise, characters up to the first non-backslash-escaped <newline>, vertical-line, or double-quote character shall be skipped and be part of the command.

  c. If the command was an **append**, **change**, or **insert** command, and the step 12.b. ended at a vertical-line character, any subsequent characters, up to the next non-backslash-escaped <newline> shall be used as input text to the command.

  d. If the command was ended by a double-quote character, all subsequent characters, up to the next non-backslash-escaped <newline>, shall be discarded.

  e. The terminating <newline> or vertical-line character shall be discarded and any subsequent characters shall be parsed as a separate *ex* command.

Command arguments shall be parsed as described by the Synopsis and Description of each individual *ex* command. This parsing shall not be <blank>-sensitive, except for the **!** argument, which must follow the command name without intervening <blank>s, and where it would otherwise be ambiguous. For example, *count* and *flag* arguments need not be <blank>-separated because `"d22p"` is not ambiguous, but *file* arguments to the *ex* **next** command must be separated by one or more <blank>s. Any <blank> in command arguments for the **abbreviate**, **unabbreviate**, **map**, and **unmap** commands can be <control>-V-escaped, in which case the <blank> shall not be used as an argument delimiter. Any <blank> in the command argument for any other command can be backslash-escaped, in which case that <blank> shall not be used as an argument delimiter.

Within command arguments for the **abbreviate**, **unabbreviate**, **map**, and **unmap** commands, any character can be <control>-V-escaped. All such escaped characters shall be treated literally and shall have no special meaning. Within command arguments for all other *ex* commands that are not regular expressions or replacement strings, any character that would otherwise have a special meaning can be backslash-escaped. Escaped characters shall be treated literally, without special meaning as shell expansion characters or `'!'`, `'%'`, and `'#'` expansion characters. See **Regular Expressions in ex** (on page 2592) and **Replacement Strings in ex** (on page 2592) for descriptions of command arguments that are regular expressions or replacement strings.

Non-backslash-escaped `'%'` characters appearing in *file* arguments to any *ex* command shall be replaced by the current pathname; unescaped `'#'` characters shall be replaced by the alternate pathname. It shall be an error if `'%'` or `'#'` characters appear unescaped in an argument and their corresponding values are not set.

Non-backslash-escaped `'!'` characters in the arguments to either the *ex* **!** command or the open and visual mode **!** command, or in the arguments to the *ex* **read** command, where the first non-<blank> after the command name is a `'!'` character, or in the arguments to the *ex* **write** command where the command name is followed by one or more <blank>s and the first non-<blank> after the command name is a `'!'` character, shall be replaced with the arguments to the last of those three commands as they appeared after all unescaped `'%'`, `'#'`, and `'!'` characters were replaced. It shall be an error if `'!'` characters appear unescaped in one of these commands and there has been no previous execution of one of these commands.

If an error occurs during the parsing or execution of an *ex* command:

- An informational message to this effect shall be written. Execution of the *ex* command shall stop, and the cursor (for example, the current line and column) shall not be further modified.
- If the *ex* command resulted from a map expansion, all characters from that map expansion shall be discarded, except as otherwise specified by the **map** command.
- Otherwise, if the *ex* command resulted from the processing of an *EXINIT* environment variable, a **.exrc** file, a **:source** command, a **–c** option, or a **+***command* specified to an *ex* **edit**, **ex**, **next**, or **visual** command, no further commands from the source of the commands shall be executed.
- Otherwise, if the *ex* command resulted from the execution of a buffer or a **global** or **v** command, no further commands caused by the execution of the buffer or the **global** or **v** command shall be executed.
- Otherwise, if the *ex* command was not terminated by a <newline>, all characters up to and including the next non-backslash-escaped <newline> shall be discarded.

Input Editing in ex

The following symbols are used in this and following sections to specify command actions.

word	In the POSIX locale, a word consists of a maximal sequence of letters, digits, and underscores, delimited at both ends by characters other than letters, digits, or underscores, or by the beginning or end of a line or the edit buffer.

When accepting input characters from the user, in either *ex* command mode or *ex* text input mode, *ex* shall enable canonical mode input processing, as defined in the System Interfaces volume of IEEE Std 1003.1-200x.

If in *ex* text input mode:

1. If the **number** edit option is set, *ex* shall prompt for input using the line number that would be assigned to the line if it is entered, in the format specified for the *ex* **number** command.
2. If the **autoindent** edit option is set, *ex* shall prompt for input using **autoindent** characters, as described by the **autoindent** edit option. **autoindent** characters shall follow the line number, if any.

If in *ex* command mode:

1. If the **prompt** edit option is set, input shall be prompted for using a single `':'` character; otherwise, there shall be no prompt.

The input characters in the following sections shall have the following effects on the input line.

Scroll

Synopsis: `eof`

See the description of the *stty eof* character in *stty*.

If in *ex* command mode:

> If the *eof* character is the first character entered on the line, the line shall be evaluated as if it contained two characters: a <control>-D and a <newline>.
>
> Otherwise, the *eof* character shall have no special meaning.

If in *ex* text input mode:

If the cursor follows an **autoindent** character, the **autoindent** characters in the line shall be modified so that a part of the next text input character will be displayed on the first column in the line after the previous **shiftwidth** edit option column boundary, and the user shall be prompted again for input for the same line.

Otherwise, if the cursor follows a `'0'`, which follows an **autoindent** character, and the `'0'` was the previous text input character, the `'0'` and all **autoindent** characters in the line shall be discarded, and the user shall be prompted again for input for the same line.

Otherwise, if the cursor follows a `'^'`, which follows an **autoindent** character, and the `'^'` was the previous text input character, the `'^'` and all **autoindent** characters in the line shall be discarded, and the user shall be prompted again for input for the same line. In addition, the **autoindent** level for the next input line shall be derived from the same line from which the **autoindent** level for the current input line was derived.

Otherwise, if there are no **autoindent** or text input characters in the line, the *eof* character shall be discarded.

Otherwise, the *eof* character shall have no special meaning.

<newline>

Synopsis: `<newline>`
`<control>-J`

If in *ex* command mode:

Cause the command line to be parsed; <control>-J shall be mapped to the <newline> for this purpose.

If in *ex* text input mode:

Terminate the current line. If there are no characters other than **autoindent** characters on the line, all characters on the line shall be discarded.

Prompt for text input on a new line after the current line. If the **autoindent** edit option is set, an appropriate number of **autoindent** characters shall be added as a prefix to the line as described by the *ex* **autoindent** edit option.

<backslash>

Synopsis: `<backslash>`

Allow the entry of a subsequent <newline> or <control>-J as a literal character, removing any special meaning that it may have to the editor during text input mode. The backslash character shall be retained and evaluated when the command line is parsed, or retained and included when the input text becomes part of the edit buffer.

<control>-V

Synopsis: `<control>-V`

Allow the entry of any subsequent character as a literal character, removing any special meaning that it may have to the editor during text input mode. The <control>-V character shall be discarded before the command line is parsed or the input text becomes part of the edit buffer.

If the ‘‘literal next’’ functionality is performed by the underlying system, it is implementation-defined whether a character other than <control>-V performs this function.

<control>-W

Synopsis: `<control>-W`

Discard the <control>-W, and the word previous to it in the input line, including any <blank>s following the word and preceding the <control>-W. If the ‘‘word erase’’ functionality is performed by the underlying system, it is implementation-defined whether a character other than <control>-W performs this function.

Command Descriptions in ex

The following symbols are used in this section to represent command modifiers. Some of these modifiers can be omitted, in which case the specified defaults shall be used.

1addr A single line address, given in any of the forms described in **Addressing in ex** (on page 2562); the default shall be the current line (`'.'`), unless otherwise specified.

If the line address is zero, it shall be an error, unless otherwise specified in the following command descriptions.

If the edit buffer is empty, and the address is specified with a command other than **=**, **append**, **insert**, **open**, **put**, **read**, or **visual**, or the address is not zero, it shall be an error.

2addr Two addresses specifying an inclusive range of lines. If no addresses are specified, the default for *2addr* shall be the current line only (`".,."`), unless otherwise specified in the following command descriptions. If one address is specified, *2addr* shall specify that line only, unless otherwise specified in the following command descriptions.

It shall be an error if the first address is greater than the second address.

If the edit buffer is empty, and the two addresses are specified with a command other than the **!**, **write**, **wq**, or **xit** commands, or either address is not zero, it shall be an error.

count A positive decimal number. If *count* is specified, it shall be equivalent to specifying an additional address to the command, unless otherwise specified by the following command descriptions. The additional address shall be equal to the last address specified to the command (either explicitly or by default) plus *count*−1.

If this would result in an address greater than the last line of the edit buffer, it shall be corrected to equal the last line of the edit buffer.

flags One or more of the characters `'+'`, `'−'`, `'#'`, `'p'`, or `'l'` (ell). The flag characters can be <blank>-separated, and in any order or combination. The characters `'#'`, `'p'`, and `'l'` shall cause lines to be written in the format specified by the **print** command with the specified *flags*.

The lines to be written are as follows:

1. All edit buffer lines written during the execution of the *ex* **&**, **~**, **list**, **number**, **open**, **print**, **s**, **visual**, and **z** commands shall be written as specified by *flags*.
2. After the completion of an *ex* command with a flag as an argument, the current line shall be written as specified by *flags*, unless the current line was the last line written by the command.

The characters `'+'` and `'−'` cause the value of the current line after the execution of the *ex* command to be adjusted by the offset address as described in **Addressing**

in ex (on page 2562). This adjustment shall occur before the current line is written as described in 2. above.

The default for *flags* shall be none.

buffer One of a number of named areas for holding text. The named buffers are specified by the alphanumeric characters of the POSIX locale. There shall also be one ‘‘unnamed’’ buffer. When no buffer is specified for editor commands that use a buffer, the unnamed buffer shall be used. Commands that store text into buffers shall store the text as it was before the command took effect, and shall store text occurring earlier in the file before text occurring later in the file, regardless of how the text region was specified. Commands that store text into buffers shall store the text into the unnamed buffer as well as any specified buffer.

In *ex* commands, buffer names are specified as the name by itself. In open or visual mode commands the name is preceded by a double quote (`'"'`) character.

If the specified buffer name is an uppercase character, and the buffer contents are to be modified, the buffer shall be appended to rather than being overwritten. If the buffer is not being modified, specifying the buffer name in lowercase and uppercase shall have identical results.

There shall also be buffers named by the numbers 1 through 9. In open and visual mode, if a region of text including characters from more than a single line is being modified by the *vi* **c** or **d** commands, the motion character associated with the **c** or **d** commands specifies that the buffer text shall be in line mode, or the commands **%**, **‘**, **/**, **?**, **(**, **)**, **N**, **n**, **{**, or **}** are used to define a region of text for the **c** or **d** commands, the contents of buffers 1 through 8 shall be moved into the buffer named by the next numerically greater value, the contents of buffer 9 shall be discarded, and the region of text shall be copied into buffer 1. This shall be in addition to copying the text into a user-specified buffer or unnamed buffer, or both. Numeric buffers can be specified as a source buffer for open and visual mode commands; however, specifying a numeric buffer as the write target of an open or visual mode command shall have unspecified results.

The text of each buffer shall have the characteristic of being in either line or character mode. Appending text to a non-empty buffer shall set the mode to match the characteristic of the text being appended. Appending text to a buffer shall cause the creation of at least one additional line in the buffer. All text stored into buffers by *ex* commands shall be in line mode. The *ex* commands that use buffers as the source of text specify individually how buffers of different modes are handled. Each open or visual mode command that uses buffers for any purpose specifies individually the mode of the text stored into the buffer and how buffers of different modes are handled.

file Command text used to derive a pathname. The default shall be the current pathname, as defined previously, in which case, if no current pathname has yet been established it shall be an error, except where specifically noted in the individual command descriptions that follow. If the command text contains any of the characters `'~'`, `'{'`, `'['`, `'*'`, `'?'`, `'$'`, ``'`'``, `'''`, `'"'`, and `'\'`, it shall be subjected to the process of ‘‘shell expansions’’, as described below; if more than a single pathname results and the command expects only one, it shall be an error.

The process of shell expansions in the editor shall be done as follows. The *ex* utility shall pass two arguments to the program named by the shell edit option; the first shall be **–c**, and the second shall be the string `"echo"` and the command text as a single argument. The standard output and standard error of that command shall replace the command text.

! A character that can be appended to the command name to modify its operation, as detailed in the individual command descriptions. With the exception of the *ex* **read**, **write**, and **!** commands, the '**!**' character shall only act as a modifier if there are no <blank>s between it and the command name.

remembered search direction
The *vi* commands **N** and **n** begin searching in a forwards or backwards direction in the edit buffer based on a remembered search direction, which is initially unset, and is set by the *ex* **global**, **v**, **s**, and **tag** commands, and the *vi* **/** and **?** commands.

Abbreviate

Synopsis: `ab[breviate][lhs rhs]`

If *lhs* and *rhs* are not specified, write the current list of abbreviations and do nothing more.

Implementations may restrict the set of characters accepted in *lhs* or *rh*, except that printable characters and <blank>s shall not be restricted. Additional restrictions shall be implementation-defined.

In both *lhs* and *rhs*, any character may be escaped with a <control>-V, in which case the character shall not be used to delimit *lhs* from *rhs*, and the escaping <control>-V shall be discarded.

In open and visual text input mode, if a non-word or <ESC> character that is not escaped by a <control>-V character is entered after a word character, a check shall be made for a set of characters matching *lhs*, in the text input entered during this command. If it is found, the effect shall be as if *rhs* was entered instead of *lhs*.

The set of characters that are checked is defined as follows:

1. If there are no characters inserted before the word and non-word or <ESC> characters that triggered the check, the set of characters shall consist of the word character.
2. If the character inserted before the word and non-word or <ESC> characters that triggered the check is a word character, the set of characters shall consist of the characters inserted immediately before the triggering characters that are word characters, plus the triggering word character.
3. If the character inserted before the word and non-word or <ESC> characters that triggered the check is not a word character, the set of characters shall consist of the characters that were inserted before the triggering characters that are neither <blank>s nor word characters, plus the triggering word character.

It is unspecified whether the *lhs* argument entered for the *ex* **abbreviate** and **unabbreviate** commands is replaced in this fashion. Regardless of whether or not the replacement occurs, the effect of the command shall be as if the replacement had not occurred.

Current line: Unchanged.

Current column: Unchanged.

Append

Synopsis: `[1addr] a[ppend][!]`

Enter text input mode; the input text shall be placed after the specified line. If line zero is specified, the text shall be placed at the beginning of the edit buffer.

This command shall be affected by the **number** and **autoindent** edit options; following the command name with `'!'` shall cause the **autoindent** edit option setting to be toggled for the duration of this command only.

Current line: Set to the last input line; if no lines were input, set to the specified line, or to the first line of the edit buffer if a line of zero was specified, or zero if the edit buffer is empty.

Current column: Set to non-<blank>.

Arguments

Synopsis: `ar[gs]`

Write the current argument list, with the current argument-list entry, if any, between `'['` and `']'` characters.

Current line: Unchanged.

Current column: Unchanged.

Change

Synopsis: `[2addr] c[hange][!][count]`

Enter *ex* text input mode; the input text shall replace the specified lines. The specified lines shall be copied into the unnamed buffer, which shall become a line mode buffer.

This command shall be affected by the **number** and **autoindent** edit options; following the command name with `'!'` shall cause the **autoindent** edit option setting to be toggled for the duration of this command only.

Current line: Set to the last input line; if no lines were input, set to the line before the first address, or to the first line of the edit buffer if there are no lines preceding the first address, or to zero if the edit buffer is empty.

Current column: Set to non-<blank>.

Change Directory

Synopsis:
```
chd[ir][!][directory]
cd[!][directory]
```

Change the current working directory to *directory*.

If no *directory* argument is specified, and the *HOME* environment variable is set to a non-null and non-empty value, *directory* shall default to the value named in the *HOME* environment variable. If the *HOME* environment variable is empty or is undefined, the default value of *directory* is implementation-defined.

If no `'!'` is appended to the command name, and the edit buffer has been modified since the last complete write, and the current pathname does not begin with a `'/'`, it shall be an error.

Current line: Unchanged.

Current column: Unchanged.

Copy

Synopsis:

```
[2addr] co[py] 1addr [flags]
[2addr] t 1addr [flags]
```

Copy the specified lines after the specified destination line; line zero specifies that the lines shall be placed at the beginning of the edit buffer.

Current line: Set to the last line copied.

Current column: Set to non-<blank>.

Delete

Synopsis:

```
[2addr] d[elete][buffer][count][flags]
```

Delete the specified lines into a buffer (defaulting to the unnamed buffer), which shall become a line-mode buffer.

Flags can immediately follow the command name; see **Command Line Parsing in ex** (on page 2563).

Current line: Set to the line following the deleted lines, or to the last line in the edit buffer if that line is past the end of the edit buffer, or to zero if the edit buffer is empty.

Current column: Set to non-<blank>.

Edit

Synopsis:

```
e[dit][!][+command][file]
ex[!][+command][file]
```

If no '!' is appended to the command name, and the edit buffer has been modified since the last complete write, it shall be an error.

If *file* is specified, replace the current contents of the edit buffer with the current contents of *file*, and set the current pathname to *file*. If *file* is not specified, replace the current contents of the edit buffer with the current contents of the file named by the current pathname. If for any reason the current contents of the file cannot be accessed, the edit buffer shall be empty.

The **+***command* option shall be <blank>-delimited; <blank>s within **+***command* can be escaped by preceding them with a backslash character. The **+***command* shall be interpreted as an *ex* command immediately after the contents of the edit buffer have been replaced and the current line and column have been set.

If the edit buffer is empty:

Current line: Set to 0.

Current column: Set to 1.

Otherwise, if executed while in *ex* command mode or if the **+***command* argument is specified:

Current line: Set to the last line of the edit buffer.

Current column: Set to non-<blank>.

Otherwise, if *file* is omitted or results in the current pathname:

Current line: Set to the first line of the edit buffer.

Current column: Set to non-<blank>.

Otherwise, if *file* is the same as the last file edited, the line and column shall be set as follows; if the file was previously edited, the line and column may be set as follows:

Current line: Set to the last value held when that file was last edited. If this value is not a valid line in the new edit buffer, set to the first line of the edit buffer.

Current column: If the current line was set to the last value held when the file was last edited, set to the last value held when the file was last edited. Otherwise, or if the last value is not a valid column in the new edit buffer, set to non-<blank>.

Otherwise:

Current line: Set to the first line of the edit buffer.

Current column: Set to non-<blank>.

File

Synopsis: `f[ile][file]`

If a *file* argument is specified, the alternate pathname shall be set to the current pathname, and the current pathname shall be set to *file*.

Write an informational message. If the file has a current pathname, it shall be included in this message; otherwise, the message shall indicate that there is no current pathname. If the edit buffer contains lines, the current line number and the number of lines in the edit buffer shall be included in this message; otherwise, the message shall indicate that the edit buffer is empty. If the edit buffer has been modified since the last complete write, this fact shall be included in this message. If the **readonly** edit option is set, this fact shall be included in this message. The message may contain other unspecified information.

Current line: Unchanged.

Current column: Unchanged.

Global

Synopsis:
```
[2addr] g[lobal] /pattern/ [commands]
[2addr] v /pattern/ [commands]
```

The optional `'!'` character after the **global** command shall be the same as executing the **v** command.

If *pattern* is empty (for example, `"//"`) or not specified, the last regular expression used in the editor command shall be used as the *pattern*. The *pattern* can be delimited by slashes (shown in the Synopsis), as well as any non-alphanumeric or non-<blank> other than backslash, vertical line, double quote, or <newline>.

If no lines are specified, the lines shall default to the entire file.

The **global** and **v** commands are logically two-pass operations. First, mark the lines within the specified lines for which the line excluding the terminating <newline> matches (**global**) or does not match (**v** or **global!**) the specified pattern. Second, execute the *ex* commands given by *commands*, with the current line (`'.'`) set to each marked line. If an error occurs during this process, or the contents of the edit buffer are replaced (for example, by the *ex* **:edit** command) an error message shall be written and no more commands resulting from the execution of this command shall be processed.

Multiple *ex* commands can be specified by entering multiple commands on a single line using a vertical line to delimit them, or one per line, by escaping each <newline> with a backslash.

If no commands are specified:

1. If in *ex* command mode, it shall be as if the **print** command were specified.
2. Otherwise, no command shall be executed.

For the **append**, **change**, and **insert** commands, the input text shall be included as part of the command, and the terminating period can be omitted if the command ends the list of commands. The **open** and **visual** commands can be specified as one of the commands, in which case each marked line shall cause the editor to enter open or visual mode. If open or visual mode is exited using the *vi* **Q** command, the current line shall be set to the next marked line, and open or visual mode reentered, until the list of marked lines is exhausted.

The **global**, **v**, and **undo** commands cannot be used in *commands*. Marked lines may be deleted by commands executed for lines occurring earlier in the file than the marked lines. In this case, no commands shall be executed for the deleted lines.

If the remembered search direction is not set, the **global** and **v** commands shall set it to forward.

The **autoprint** and **autoindent** edit options shall be inhibited for the duration of the **g** or **v** command.

Current line: If no commands executed, set to the last marked line. Otherwise, as specified for the executed *ex* commands.

Current column: If no commands are executed, set to non-<blank>; otherwise, as specified for the individual *ex* commands.

Insert

Synopsis: `[1addr] i[nsert][!]`

Enter *ex* text input mode; the input text shall be placed before the specified line. If the line is zero or 1, the text shall be placed at the beginning of the edit buffer.

This command shall be affected by the **number** and **autoindent** edit options; following the command name with `'!'` shall cause the **autoindent** edit option setting to be toggled for the duration of this command only.

Current line: Set to the last input line; if no lines were input, set to the line before the specified line, or to the first line of the edit buffer if there are no lines preceding the specified line, or zero if the edit buffer is empty.

Current column: Set to non-<blank>.

Join

Synopsis: `[2addr] j[oin][!][count][flags]`

If *count* is specified:

If no address was specified, the **join** command shall behave as if *2addr* were the current line and the current line plus *count* (.,. + *count*).

If one address was specified, the **join** command shall behave as if *2addr* were the specified address and the specified address plus *count* (*addr*,*addr* + *count*).

If two addresses were specified, the **join** command shall behave as if an additional address, equal to the last address plus *count* −1 (*addr1*,*addr2*,*addr2* + *count* −1), was specified.

If this would result in a second address greater than the last line of the edit buffer, it shall be corrected to be equal to the last line of the edit buffer.

If no *count* is specified:

If no address was specified, the **join** command shall behave as if *2addr* were the current line and the next line (.,. +1).

If one address was specified, the **join** command shall behave as if *2addr* were the specified address and the next line (*addr*,*addr* +1).

Join the text from the specified lines together into a single line, which shall replace the specified lines.

If a `'!'` character is appended to the command name, the **join** shall be without modification of any line, independent of the current locale.

Otherwise, in the POSIX locale, set the current line to the first of the specified lines, and then, for each subsequent line, proceed as follows:

1. Discard leading <space>s from the line to be joined.
2. If the line to be joined is now empty, delete it, and skip steps 3 through 5.
3. If the current line ends in a <blank>, or the first character of the line to be joined is a `')'` character, join the lines without further modification.
4. If the last character of the current line is a `'.'`, join the lines with two <space>s between them.
5. Otherwise, join the lines with a single <space> between them.

Current line: Set to the first line specified.

Current column: Set to non-<blank>.

List

Synopsis: **[***2addr***]** l**[**ist**][***count***][***flags***]**

This command shall be equivalent to the *ex* command:

[*2addr***]** p**[**rint**][***count***]** l**[***flags***]**

See **Print** (on page 2580).

Map

Synopsis: `map[!][lhs rhs]`

If *lhs* and *rhs* are not specified:

1. If `'!'` is specified, write the current list of text input mode maps.
2. Otherwise, write the current list of command mode maps.
3. Do nothing more.

Implementations may restrict the set of characters accepted in *lhs* or *rhs*, except that printable characters and <blank>s shall not be restricted. Additional restrictions shall be implementation-defined. In both *lhs* and *rhs*, any character can be escaped with a <control>-V, in which case the character shall not be used to delimit *lhs* from *rhs*, and the escaping <control>-V shall be discarded.

If the character `'!'` is appended to the **map** command name, the mapping shall be effective during open or visual text input mode rather than **open** or **visual** command mode. This allows *lhs* to have two different **map** definitions at the same time: one for command mode and one for text input mode.

For command mode mappings:

> When the *lhs* is entered as any part of a *vi* command in open or visual mode (but not as part of the arguments to the command), the action shall be as if the corresponding *rhs* had been entered.
>
> If any character in the command, other than the first, is escaped using a <control>-V character, that character shall not be part of a match to an *lhs*.
>
> It is unspecified whether implementations shall support **map** commands where the *lhs* is more than a single character in length, where the first character of the *lhs* is printable.
>
> If *lhs* contains more than one character and the first character is `'#'`, followed by a sequence of digits corresponding to a numbered function key, then when this function key is typed it shall be mapped to *rhs*. Characters other than digits following a `'#'` character also represent the function key named by the characters in the *lhs* following the `'#'` and may be mapped to *rhs*. It is unspecified how function keys are named or what function keys are supported.

For text input mode mappings:

> When the *lhs* is entered as any part of text entered in open or visual text input modes, the action shall be as if the corresponding *rhs* had been entered.
>
> If any character in the input text is escaped using a <control>-V character, that character shall not be part of a match to an *lhs*.
>
> It is unspecified whether the *lhs* argument entered for the **map** or **unmap** commands is replaced in this fashion. Regardless of whether or not the replacement occurs, the effect of the command shall be as if the replacement had not occurred.

If only part of the *lhs* is entered, it is unspecified how long the editor will wait for additional, possibly matching characters before treating the already entered characters as not matching the *lhs*.

The *rhs* characters shall themselves be subject to remapping, unless otherwise specified by the **remap** edit option, except that if the characters in *lhs* occur as prefix characters in *rhs*, those characters shall not be remapped.

On block-mode terminals, the mapping need not occur immediately (for example, it may occur after the terminal transmits a group of characters to the system), but it shall achieve the same results as if it occurred immediately.

Current line: Unchanged.

Current column: Unchanged.

Mark

Synopsis:

```
[1addr] ma[rk] character
[1addr] k character
```

Implementations shall support *character* values of a single lowercase letter of the POSIX locale and the characters `'‘'` and `'’'`; support of other characters is implementation-defined.

If executing the *vi* **m** command, set the specified mark to the current line and 1-based numbered character referenced by the current column, if any; otherwise, column position 1.

Otherwise, set the specified mark to the specified line and 1-based numbered first non-<blank> non- <newline> in the line, if any; otherwise, the last non-<newline> in the line, if any; otherwise, column position 1.

The mark shall remain associated with the line until the mark is reset or the line is deleted. If a deleted line is restored by a subsequent **undo** command, any marks previously associated with the line, which have not been reset, shall be restored as well. Any use of a mark not associated with a current line in the edit buffer shall be an error.

The marks **‘** and **’** shall be set as described previously, immediately before the following events occur in the editor:

1. The use of `'$'` as an *ex* address
2. The use of a positive decimal number as an *ex* address
3. The use of a search command as an *ex* address
4. The use of a mark reference as an *ex* address
5. The use of the following open and visual mode commands: <control>-], **%**, **(**, **)**, **[**, **]**, **{**, **}**.
6. The use of the following open and visual mode commands: **’**, **G**, **H**, **L**, **M**, **z** if the current line will change as a result of the command
7. The use of the open and visual mode commands: **/**, **?**, **N**, **‘**, **n** if the current line or column will change as a result of the command
8. The use of the *ex* mode commands: **z**, **undo**, **global**, **v**

For rules 1., 2., 3., and 4., the **‘** and **’** marks shall not be set if the *ex* command is parsed as specified by rule 6.a. in **Command Line Parsing in ex** (on page 2563).

For rules 5., 6., and 7., the **‘** and **’** marks shall not be set if the commands are used as motion commands in open and visual mode.

For rules 1., 2., 3., 4., 5., 6., 7., and 8., the **‘** and **’** marks shall not be set if the command fails.

The **‘** and **’** marks shall be set as described previously, each time the contents of the edit buffer are replaced (including the editing of the initial buffer), if in open or visual mode, or if in **ex** mode and the edit buffer is not empty, before any commands or movements (including commands or movements specified by the −**c** or −**t** options or the **+***command* argument) are executed on the edit buffer. If in open or visual mode, the marks shall be set as if executing the *vi*

m command; otherwise, as if executing the *ex* **mark** command.

When changing from **ex** mode to open or visual mode, if the **‘** and **’** marks are not already set, the **‘** and **’** marks shall be set as described previously.

Current line: Unchanged.

Current column: Unchanged.

Move

Synopsis: `[2addr] m[ove] 1addr [flags]`

Move the specified lines after the specified destination line. A destination of line zero specifies that the lines shall be placed at the beginning of the edit buffer. It shall be an error if the destination line is within the range of lines to be moved.

Current line: Set to the last of the moved lines.

Current column: Set to non-<blank>.

Next

Synopsis: `n[ext][!][+command][file ...]`

If no `'!'` is appended to the command name, and the edit buffer has been modified since the last complete write, it shall be an error, unless the file is successfully written as specified by the **autowrite** option.

If one or more files is specified:

1. Set the argument list to the specified filenames.
2. Set the current argument list reference to be the first entry in the argument list.
3. Set the current pathname to the first filename specified.

Otherwise:

1. It shall be an error if there are no more filenames in the argument list after the filename currently referenced.
2. Set the current pathname and the current argument list reference to the filename after the filename currently referenced in the argument list.

Replace the contents of the edit buffer with the contents of the file named by the current pathname. If for any reason the contents of the file cannot be accessed, the edit buffer shall be empty.

This command shall be affected by the **autowrite** and **writeany** edit options.

The **+***command* option shall be <blank>-delimited; <blank>s can be escaped by preceding them with a backslash character. The **+***command* shall be interpreted as an *ex* command immediately after the contents of the edit buffer have been replaced and the current line and column have been set.

Current line: Set as described for the **edit** command.

Current column: Set as described for the **edit** command.

Number

Synopsis: `[2addr] nu[mber][count][flags]`
`[2addr] #[count][flags]`

These commands shall be equivalent to the *ex* command:

`[2addr] p[rint][count] #[flags]`

See **Print**.

Open

Synopsis: `[1addr] o[pen] /pattern/ [flags]`

This command need not be supported on block-mode terminals or terminals with insufficient capabilities. If standard input, standard output, or standard error are not terminal devices, the results are unspecified.

Enter open mode.

The trailing delimiter can be omitted from pattern at the end of the command line. If pattern is empty (for example, `"//"`) or not specified, the last regular expression used in the editor shall be used as the pattern. The pattern can be delimited by slashes (shown in the Synopsis), as well as any alphanumeric, or non-<blank> other than backslash, vertical line, double quote, or <newline>.

Current line: Set to the specified line.

Current column: Set to non-<blank>.

Preserve

Synopsis: `pre[serve]`

Save the edit buffer in a form that can later be recovered by using the **–r** option or by using the *ex* **recover** command. After the file has been preserved, a mail message shall be sent to the user. This message shall be readable by invoking the *mailx* utility. The message shall contain the name of the file, the time of preservation, and an *ex* command that could be used to recover the file. Additional information may be included in the mail message.

Current line: Unchanged.

Current column: Unchanged.

Print

Synopsis: `[2addr] p[rint][count][flags]`

Write the addressed lines. The behavior is unspecified if the number of columns on the display is less than the number of columns required to write any single character in the lines being written.

Non-printable characters, except for the <tab>, shall be written as implementation-defined multi-character sequences.

If the **#** flag is specified or the **number** edit option is set, each line shall be preceded by its line number in the following format:

`"%6dΔΔ", <line number>`

If the **l** flag is specified or the **list** edit option is set:

1. The characters listed in the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions shall be written as the corresponding escape sequence.
2. Non-printable characters not in the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions shall be written as one three-digit octal number (with a preceding backslash) for each byte in the character (most significant byte first). If the size of a byte on the system is greater than 9 bits, the format used for non-printable characters is implementation-defined.
3. The end of each line shall be marked with a `'$'`, and literal `'$'` characters within the line shall be written with a preceding backslash.

Long lines shall be folded; the length at which folding occurs is unspecified, but should be appropriate for the output terminal, considering the number of columns of the terminal.

If a line is folded, and the **l** flag is not specified and the **list** edit option is not set, it is unspecified whether a multi-column character at the folding position is separated; it shall not be discarded.

Current line: Set to the last written line.

Current column: Unchanged if the current line is unchanged; otherwise, set to non-<blank>.

Put

Synopsis: **[***1addr***]** `pu`**[**`t`**][***buffer***]**

Append text from the specified buffer (by default, the unnamed buffer) to the specified line; line zero specifies that the text shall be placed at the beginning of the edit buffer. Each portion of a line in the buffer shall become a new line in the edit buffer, regardless of the mode of the buffer.

Current line: Set to the last line entered into the edit buffer.

Current column: Set to non-<blank>.

Quit

Synopsis: `q`**[**`uit`**][**`!`**]**

If no `'!'` is appended to the command name:

1. If the edit buffer has been modified since the last complete write, it shall be an error.
2. If there are filenames in the argument list after the filename currently referenced, and the last command was not a **quit**, **wq**, **xit**, or **ZZ** (see **Exit** (on page 3220)) command, it shall be an error.

Otherwise, terminate the editing session.

Read

Synopsis: **[***1addr***]** `r`**[**`ead`**][**`!`**][***file***]**

If `'!'` is not the first non-<blank> to follow the command name, a copy of the specified file shall be appended into the edit buffer after the specified line; line zero specifies that the copy shall be placed at the beginning of the edit buffer. The number of lines and bytes read shall be written. If no *file* is named, the current pathname shall be the default. If there is no current pathname, then *file* shall become the current pathname. If there is no current pathname or *file* operand, it shall be an error. Specifying a *file* that is not of type regular shall have unspecified results.

Otherwise, if *file* is preceded by `'!'`, the rest of the line after the `'!'` shall have `'%'`, `'#'`, and `'!'` characters expanded as described in **Command Line Parsing in ex** (on page 2563).

The *ex* utility shall then pass two arguments to the program named by the *shell* edit option; the first shall be **–c** and the second shall be the expanded arguments to the **read** command as a single argument. The standard input of the program shall be set to the standard input of the *ex* program when it was invoked. The standard error and standard output of the program shall be appended into the edit buffer after the specified line.

Each line in the copied file or program output (as delimited by <newline>s or the end of the file or output if it is not immediately preceded by a <newline>), shall be a separate line in the edit buffer. Any occurrences of <carriage-return> and <newline> pairs in the output shall be treated as single <newline>s.

The special meaning of the `'!'` following the **read** command can be overridden by escaping it with a backslash character.

Current line: If no lines are added to the edit buffer, unchanged. Otherwise, if in open or visual mode, set to the first line entered into the edit buffer. Otherwise, set to the last line entered into the edit buffer.

Current column: Set to non-<blank>.

Recover

Synopsis: `rec[over][!]` *`file`*

If no `'!'` is appended to the command name, and the edit buffer has been modified since the last complete write, it shall be an error.

If no *file* operand is specified, then the current pathname shall be used. If there is no current pathname or *file* operand, it shall be an error.

If no recovery information has previously been saved about *file*, the **recover** command shall behave identically to the **edit** command, and an informational message to this effect shall be written.

Otherwise, set the current pathname to *file*, and replace the current contents of the edit buffer with the recovered contents of *file*. If there are multiple instances of the file to be recovered, the one most recently saved shall be recovered, and an informational message that there are previous versions of the file that can be recovered shall be written. The editor shall behave as if the contents of the edit buffer have already been modified.

Current file: Set as described for the **edit** command.

Current column: Set as described for the **edit** command.

Rewind

Synopsis: `rew[ind][!]`

If no `'!'` is appended to the command name, and the edit buffer has been modified since the last complete write, it shall be an error, unless the file is successfully written as specified by the **autowrite** option.

If the argument list is empty, it shall be an error.

The current argument list reference and the current pathname shall be set to the first filename in the argument list.

Replace the contents of the edit buffer with the contents of the file named by the current pathname. If for any reason the contents of the file cannot be accessed, the edit buffer shall be empty.

This command shall be affected by the **autowrite** and **writeany** edit options.

Current line: Set as described for the **edit** command.

Current column: Set as described for the **edit** command.

Set

Synopsis: `se[t][option[=[value]] ...][nooption ...][option? ...][all]`

When no arguments are specified, write the value of the **term** edit option and those options whose values have been changed from the default settings; when the argument *all* is specified, write all of the option values.

Giving an option name followed by the character `'?'` shall cause the current value of that option to be written. The `'?'` can be separated from the option name by zero or more <blank>s. The `'?'` shall be necessary only for Boolean valued options. Boolean options can be given values by the form **set** *option* to turn them on or **set no***option* to turn them off; string and numeric options can be assigned by the form **set** *option*=*value*. Any <blank>s in strings can be included as is by preceding each <blank> with an escaping backslash. More than one option can be set or listed by a single set command by specifying multiple arguments, each separated from the next by one or more <blank>s.

See **Edit Options in ex** (on page 2593) for details about specific options.

Current line: Unchanged.

Current column: Unchanged.

Shell

Synopsis: `sh[ell]`

Invoke the program named in the **shell** edit option with the single argument **−i** (interactive mode). Editing shall be resumed when the program exits.

Current line: Unchanged.

Current column: Unchanged.

Source

Synopsis: `so[urce] file`

Read and execute *ex* commands from *file*. Lines in the file that are blank lines shall be ignored.

Current line: As specified for the individual *ex* commands.

Current column: As specified for the individual *ex* commands.

Substitute

Synopsis:

```
[2addr] s[ubstitute][/pattern/repl/[options][count][flags]]
[2addr] &[options][count][flags]]
[2addr] ~[options][count][flags]]
```

Replace the first instance of the pattern *pattern* by the string *repl* on each specified line. (See **Regular Expressions in ex** (on page 2592) and **Replacement Strings in ex** (on page 2592).) Any non-alphabetic, non-<blank> delimiter other than `'\\'`, `'|'`, double quote, or <newline> can be used instead of `'/'`. Backslash characters can be used to escape delimiters, backslash characters, and other special characters.

The trailing delimiter can be omitted from *pattern* or from *repl* at the end of the command line. If both *pattern* and *repl* are not specified or are empty (for example, `"//"`), the last **s** command shall be repeated. If only *pattern* is not specified or is empty, the last regular expression used in the editor shall be used as the pattern. If only *repl* is not specified or is empty, the pattern shall be replaced by nothing. If the entire replacement pattern is `'%'`, the last replacement pattern to an **s** command shall be used.

Entering a <carriage-return> in *repl* (which requires an escaping backslash in *ex* mode and an escaping <control>-V in open or *vi* mode) shall split the line at that point, creating a new line in the edit buffer. The <carriage-return> shall be discarded.

If options include the letter `'g'` (**global**), all non-overlapping instances of the pattern in the line shall be replaced.

If options includes the letter `'c'` (**confirm**), then before each substitution the line shall be written; the written line shall reflect all previous substitutions. On the following line, <space>s shall be written beneath the characters from the line that are before the *pattern* to be replaced, and `'^'` characters written beneath the characters included in the *pattern* to be replaced. The *ex* utility shall then wait for a response from the user. An affirmative response shall cause the substitution to be done, while any other input shall not make the substitution. An affirmative response shall consist of a line with the affirmative response (as defined by the current locale) at the beginning of the line. This line shall be subject to editing in the same way as the *ex* command line.

If interrupted (see the ASYNCHRONOUS EVENTS section), any modifications confirmed by the user shall be preserved in the edit buffer after the interrupt.

If the remembered search direction is not set, the **s** command shall set it to forward.

In the second Synopsis, the **&** command shall repeat the previous substitution, as if the **&** command were replaced by:

```
s/pattern/repl/
```

where *pattern* and *repl* are as specified in the previous **s**, **&**, or **~** command.

In the third Synopsis, the **~** command shall repeat the previous substitution, as if the `'~'` were replaced by:

```
s/pattern/repl/
```

where *pattern* shall be the last regular expression specified to the editor, and *repl* shall be from the previous substitution (including **&** and **~**) command.

These commands shall be affected by the *LC_MESSAGES* environment variable.

Current line: Set to the last line in which a substitution occurred, or, unchanged if no substitution occurred.

Current column: Set to non-<blank>.

Suspend

Synopsis:
```
su[spend][!]
st[op][!]
```

Allow control to return to the invoking process; *ex* shall suspend itself as if it had received the SIGTSTP signal. The suspension shall occur only if job control is enabled in the invoking shell (see the description of *set* −**m**).

These commands shall be affected by the **autowrite** and **writeany** edit options.

The current **susp** character (see *stty*) shall be equivalent to the **suspend** command.

Tag

Synopsis:
```
ta[g][!] tagstring
```

The results are unspecified if the format of a tags file is not as specified by the *ctags* utility (see *ctags*) description.

The **tag** command shall search for *tagstring* in the tag files referred to by the **tag** edit option, in the order they are specified, until a reference to *tagstring* is found. Files shall be searched from beginning to end. If no reference is found, it shall be an error and an error message to this effect shall be written. If the reference is not found, or if an error occurs while processing a file referred to in the **tag** edit option, it shall be an error, and an error message shall be written at the first occurrence of such an error.

Otherwise, if the tags file contained a pattern, the pattern shall be treated as a regular expression used in the editor; for example, for the purposes of the **s** command.

If the *tagstring* is in a file with a different name than the current pathname, set the current pathname to the name of that file, and replace the contents of the edit buffer with the contents of that file. In this case, if no `'!'` is appended to the command name, and the edit buffer has been modified since the last complete write, it shall be an error, unless the file is successfully written as specified by the **autowrite** option.

This command shall be affected by the **autowrite**, **tag**, **taglength**, and **writeany** edit options.

Current line: If the tags file contained a line number, set to that line number. If the line number is larger than the last line in the edit buffer, an error message shall be written and the current line shall be set as specified for the **edit** command.

If the tags file contained a pattern, set to the first occurrence of the pattern in the file. If no matching pattern is found, an error message shall be written and the current line shall be set as specified for the **edit** command.

Current column: If the tags file contained a line-number reference and that line-number was not larger than the last line in the edit buffer, or if the tags file contained a pattern and that pattern was found, set to non-<blank>. Otherwise, set as specified for the **edit** command.

Unabbreviate

Synopsis: `una[bbrev] lhs`

If *lhs* is not an entry in the current list of abbreviations (see **Abbreviate** (on page 2571)), it shall be an error. Otherwise, delete *lhs* from the list of abbreviations.

Current line: Unchanged.

Current column: Unchanged.

Undo

Synopsis: `u[ndo]`

Reverse the changes made by the last command that modified the contents of the edit buffer, including **undo**. For this purpose, the **global**, **v**, **open**, and **visual** commands, and commands resulting from buffer executions and mapped character expansions, are considered single commands.

If no action that can be undone preceded the **undo** command, it shall be an error.

If the **undo** command restores lines that were marked, the mark shall also be restored unless it was reset subsequent to the deletion of the lines.

Current line:

1. If lines are added or changed in the file, set to the first line added or changed.
2. Set to the line before the first line deleted, if it exists.
3. Set to 1 if the edit buffer is not empty.
4. Set to zero.

Current column: Set to non-<blank>.

Unmap

Synopsis: `unm[ap][!] lhs`

If `'!'` is appended to the command name, and if *lhs* is not an entry in the list of text input mode map definitions, it shall be an error. Otherwise, delete *lhs* from the list of text input mode map definitions.

If no `'!'` is appended to the command name, and if *lhs* is not an entry in the list of command mode map definitions, it shall be an error. Otherwise, delete *lhs* from the list of command mode map definitions.

Current line: Unchanged.

Current column: Unchanged.

Version

Synopsis: `ve[rsion]`

Write a message containing version information for the editor. The format of the message is unspecified.

Current line: Unchanged.

Current column: Unchanged.

Visual

Synopsis: `[1addr] vi[sual][type][count][flags]`

If *ex* is currently in open or visual mode, the Synopsis and behavior of the visual command shall be the same as the **edit** command, as specified by **Edit** (on page 2573).

Otherwise, this command need not be supported on block-mode terminals or terminals with insufficient capabilities. If standard input, standard output, or standard error are not terminal devices, the results are unspecified.

If *count* is specified, the value of the **window** edit option shall be set to *count* (as described in **window** (on page 2599)). If the `'^'` type character was also specified, the **window** edit option shall be set before being used by the type character.

Enter visual mode. If *type* is not specified, it shall be as if a *type* of `'+'` was specified. The *type* shall cause the following effects:

`+` Place the beginning of the specified line at the top of the display.

`-` Place the end of the specified line at the bottom of the display.

`.` Place the beginning of the specified line in the middle of the display.

`^` If the specified line is less than or equal to the value of the **window** edit option, set the line to 1; otherwise, decrement the line by the value of the **window** edit option minus 1. Place the beginning of this line as close to the bottom of the displayed lines as possible, while still displaying the value of the **window** edit option number of lines.

Current line: Set to the specified line.

Current column: Set to non-<blank>.

Write

Synopsis:
```
[2addr] w[rite][!][>>][file]
[2addr] w[rite][!][file]
[2addr] wq[!][>>][file]
```

If no lines are specified, the lines shall default to the entire file.

The command **wq** shall be equivalent to a **write** command followed by a **quit** command; **wq!** shall be equivalent to **write!** followed by **quit**. In both cases, if the **write** command fails, the **quit** shall not be attempted.

If the command name is not followed by one or more <blank>s, or *file* is not preceded by a `'!'` character, the **write** shall be to a file.

1. If the **>>** argument is specified, and the file already exists, the lines shall be appended to the file instead of replacing its contents. If the **>>** argument is specified, and the file does not already exist, it is unspecified whether the write shall proceed as if the **>>** argument had not been specified or if the write shall fail.
2. If the **readonly** edit option is set (see **readonly** (on page 2596)), the **write** shall fail.
3. If *file* is specified, and is not the current pathname, and the file exists, the **write** shall fail.
4. If *file* is not specified, the current pathname shall be used. If there is no current pathname, the **write** command shall fail.
5. If the current pathname is used, and the current pathname has been changed by the **file** or **read** commands, and the file exists, the **write** shall fail. If the **write** is successful,

subsequent **write**s shall not fail for this reason (unless the current pathname is changed again).

6. If the whole edit buffer is not being written, and the file to be written exists, the **write** shall fail.

For rules 1., 2., 4., and 5., the **write** can be forced by appending the character `'!'` to the command name.

For rules 2., 4., and 5., the **write** can be forced by setting the **writeany** edit option.

Additional, implementation-defined tests may cause the **write** to fail.

If the edit buffer is empty, a file without any contents shall be written.

An informational message shall be written noting the number of lines and bytes written.

Otherwise, if the command is followed by one or more <blank>s, and the file is preceded by `'!'`, the rest of the line after the `'!'` shall have `'%'`, `'#'`, and `'!'` characters expanded as described in **Command Line Parsing in ex** (on page 2563).

The *ex* utility shall then pass two arguments to the program named by the **shell** edit option; the first shall be **–c** and the second shall be the expanded arguments to the **write** command as a single argument. The specified lines shall be written to the standard input of the command. The standard error and standard output of the program. if any, shall be written as described for the **print** command. If the last character in that output is not a <newline>, a <newline> shall be written at the end of the output.

The special meaning of the `'!'` following the **write** command can be overridden by escaping it with a backslash character.

Current line: Unchanged.

Current column: Unchanged.

Write and Exit

Synopsis: `[2addr] x[it][!][file]`

If the edit buffer has not been modified since the last complete **write**, **xit** shall be equivalent to the **quit** command, or if a `'!'` is appended to the command name, to **quit!**.

Otherwise, **xit** shall be equivalent to the **wq** command, or if a `'!'` is appended to the command name, to **wq!**.

Current line: Unchanged.

Current column: Unchanged.

Yank

Synopsis: `[2addr] ya[nk][buffer][count]`

Copy the specified lines to the specified buffer (by default, the unnamed buffer), which shall become a line-mode buffer.

Current line: Unchanged.

Current column: Unchanged.

Adjust Window

Synopsis: `[1addr] z[!][type ...][count][flags]`

If no line is specified, the current line shall be the default; if *type* is omitted as well, the current line value shall first be incremented by 1. If incrementing the current line would cause it to be greater than the last line in the edit buffer, it shall be an error.

If there are <blank>s between the *type* argument and the preceding **z** command name or optional `'!'` character, it shall be an error.

If *count* is specified, the value of the **window** edit option shall be set to *count* (as described in **window** (on page 2599)). If *count* is omitted, it shall default to 2 times the value of the **scroll** edit option, or if **!** was specified, the number of lines in the display minus 1.

If *type* is omitted, then *count* lines starting with the specified line shall be written. Otherwise, *count* lines starting with the line specified by the *type* argument shall be written.

The *type* argument shall change the lines to be written. The possible values of *type* are as follows:

– The specified line shall be decremented by the following value:

```
(((number of ''−'' characters) x count) −1)
```

If the calculation would result in a number less than 1, it shall be an error. Write lines from the edit buffer, starting at the new value of line, until *count* lines or the last line in the edit buffer has been written.

\+ The specified line shall be incremented by the following value:

```
(((number of ''+'' characters) −1) x count) +1
```

If the calculation would result in a number greater than the last line in the edit buffer, it shall be an error. Write lines from the edit buffer, starting at the new value of line, until *count* lines or the last line in the edit buffer has been written.

=,. If more than a single `'.'` or `'='` is specified, it shall be an error. The following steps shall be taken:

1. If *count* is zero, nothing shall be written.
2. Write as many of the *N* lines before the current line in the edit buffer as exist. If *count* or `'!'` was specified, *N* shall be:

   ```
   (count −1) /2
   ```

   Otherwise, *N* shall be:

   ```
   (count −3) /2
   ```

   If *N* is a number less than 3, no lines shall be written.
3. If `'='` was specified as the type character, write a line consisting of the smaller of the number of columns in the display divided by two, or 40 `'−'` characters.
4. Write the current line.
5. Repeat step 3.
6. Write as many of the *N* lines after the current line in the edit buffer as exist. *N* shall be defined as in step 2. If *N* is a number less than 3, no lines shall be written. current line in the edit buffer as exist. If count is less than 3, no lines shall be written.

^ The specified line shall be decremented by the following value:

`(((number of ''^'' characters) +1) x count) −1`

If the calculation would result in a number less than 1, it shall be an error. Write lines from the edit buffer, starting at the new value of line, until *count* lines or the last line in the edit buffer has been written.

Current line: Set to the last line written, unless the type is **=**, in which case, set to the specified line.

Current column: Set to non-<blank>.

Escape

Synopsis: `!` *command*
[*addr***]**`!` *command*

The contents of the line after the `'!'` shall have `'%'`, `'#'`, and `'!'` characters expanded as described in **Command Line Parsing in ex** (on page 2563). If the expansion causes the text of the line to change, it shall be redisplayed, preceded by a single `'!'` character.

The *ex* utility shall execute the program named by the **shell** edit option. It shall pass two arguments to the program; the first shall be **−c**, and the second shall be the expanded arguments to the **!** command as a single argument.

If no lines are specified, the standard input, standard output, and standard error of the program shall be set to the standard input, standard output, and standard error of the *ex* program when it was invoked. In addition, a warning message shall be written if the edit buffer has been modified since the last complete write, and the **warn** edit option is set.

If lines are specified, they shall be passed to the program as standard input, and the standard output and standard error of the program shall replace those lines in the edit buffer. Each line in the program output (as delimited by <newline>s or the end of the output if it is not immediately preceded by a <newline>), shall be a separate line in the edit buffer. Any occurrences of <carriage-return> and <newline> pairs in the output shall be treated as single <newline>s. The specified lines shall be copied into the unnamed buffer before they are replaced, and the unnamed buffer shall become a line-mode buffer.

If in *ex* mode, a single `'!'` character shall be written when the program completes.

This command shall be affected by the **shell** and **warn** edit options. If no lines are specified, this command shall be affected by the **autowrite** and **writeany** edit options. If lines are specified, this command shall be affected by the **autoprint** edit option.

Current line:

1. If no lines are specified, unchanged.
2. Otherwise, set to the last line read in, if any lines are read in.
3. Otherwise, set to the line before the first line of the lines specified, if that line exists.
4. Otherwise, set to the first line of the edit buffer if the edit buffer is not empty.
5. Otherwise, set to zero.

Current column: If no lines are specified, unchanged. Otherwise, set to non-<blank>.

Shift Left

Synopsis: `[2addr] <[< ...][count][flags]`

Shift the specified lines to the start of the line; the number of column positions to be shifted shall be the number of command characters times the value of the **shiftwidth** edit option. Only leading <blank>s shall be deleted or changed into other <blank>s in shifting; other characters shall not be affected.

Lines to be shifted shall be copied into the unnamed buffer, which shall become a line-mode buffer.

This command shall be affected by the **autoprint** edit option.

Current line: Set to the last line in the lines specified.

Current column: Set to non-<blank>.

Shift Right

Synopsis: `[2addr] >[> ...][count][flags]`

Shift the specified lines away from the start of the line; the number of column positions to be shifted shall be the number of command characters times the value of the **shiftwidth** edit option. The shift shall be accomplished by adding <blank>s as a prefix to the line or changing leading <blank>s into other <blank>s. Empty lines shall not be changed.

Lines to be shifted shall be copied into the unnamed buffer, which shall become a line-mode buffer.

This command shall be affected by the **autoprint** edit option.

Current line: Set to the last line in the lines specified.

Current column: Set to non-<blank>.

<control>-D

Synopsis: `<control>-D`

Write the next *n* lines, where *n* is the minimum of the values of the **scroll** edit option and the number of lines after the current line in the edit buffer. If the current line is the last line of the edit buffer it shall be an error.

Current line: Set to the last line written.

Current column: Set to non-<blank>.

Write Line Number

Synopsis: `[1addr] = [flags]`

If *line* is not specified, it shall default to the last line in the edit buffer. Write the line number of the specified line.

Current line: Unchanged.

Current column: Unchanged.

Execute

Synopsis: `[2addr] @ buffer`
`[2addr] * buffer`

If no buffer is specified or is specified as '@' or '*', the last buffer executed shall be used. If no previous buffer has been executed, it shall be an error.

For each line specified by the addresses, set the current line ('.') to the specified line, and execute the contents of the named *buffer* (as they were at the time the **@** command was executed) as *ex* commands. For each line of a line-mode buffer, and all but the last line of a character-mode buffer, the *ex* command parser shall behave as if the line was terminated by a <newline>.

If an error occurs during this process, or a line specified by the addresses does not exist when the current line would be set to it, or more than a single line was specified by the addresses, and the contents of the edit buffer are replaced (for example, by the *ex* **:edit** command) an error message shall be written, and no more commands resulting from the execution of this command shall be processed.

Current line: As specified for the individual *ex* commands.

Current column: As specified for the individual *ex* commands.

Regular Expressions in ex

The *ex* utility shall support regular expressions that are a superset of the basic regular expressions described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions. A null regular expression ("//") shall be equivalent to the last regular expression encountered.

Regular expressions can be used in addresses to specify lines and, in some commands (for example, the **substitute** command), to specify portions of a line to be substituted.

The following constructs can be used to enhance the basic regular expressions:

\< Match the beginning of a *word*. (See the definition of *word* at the beginning of **Command Descriptions in ex** (on page 2569).)

\> Match the end of a *word*.

~ Match the replacement part of the last **substitute** command. The tilde ('~') character can be escaped in a regular expression to become a normal character with no special meaning. The backslash shall be discarded.

When the editor option **magic** is not set, the only characters with special meanings shall be '^' at the beginning of a pattern, '$' at the end of a pattern, and '\'. The characters '.', '*', '[', and '~' shall be treated as ordinary characters unless preceded by a '\'; when preceded by a '\' they shall regain their special meaning, or in the case of backslash, be handled as a single backslash. Backslashes used to escape other characters shall be discarded.

Replacement Strings in ex

The character '&' ('\&' if the editor option **magic** is not set) in the replacement string shall stand for the text matched by the pattern to be replaced. The character '~' ('\~' if **magic** is not set) shall be replaced by the replacement part of the previous **substitute** command. The sequence '\n', where *n* is an integer, shall be replaced by the text matched by the pattern enclosed in the *n*th set of parentheses '\(' and '\)'.

The strings '\l', '\u', '\L', and '\U' can be used to modify the case of elements in the replacement string (using the '\&' or "\"digit) notation. The string '\l' ('\u') shall cause the character that follows to be converted to lowercase (uppercase). The string `'\L'` (`'\U'`) shall cause all characters subsequent to it to be converted to lowercase (uppercase) as they are inserted by the substitution until the string `'\e'` or `'\E'`, or the end of the replacement string, is encountered.

Otherwise, any character following a backslash shall be treated as that literal character, and the escaping backslash shall be discarded.

An example of case conversion with the **s** command is as follows:

```
:p
The cat sat on the mat.
:s/\<.at\>/\u&/gp
The Cat Sat on the Mat.
:s/S\(.*\)M/S\U\1\eM/p
The Cat SAT ON THE Mat.
```

Edit Options in ex

The *ex* utility has a number of options that modify its behavior. These options have default settings, which can be changed using the **set** command.

Options are Boolean unless otherwise specified.

autoindent, ai

[Default *unset*]

If **autoindent** is set, each line in input mode shall be indented (using first as many <tab>s as possible, as determined by the editor option **tabstop**, and then using <space>s) to align with another line, as follows:

1. If in open or visual mode and the text input is part of a line-oriented command (see the EXTENDED DESCRIPTION in *vi*), align to the first column. Otherwise, if in open or visual mode, indentation for each line shall be set as follows:
   a. If a line was previously inserted as part of this command, it shall be set to the indentation of the last inserted line by default, or as otherwise specified for the <control>-D character in **Input Mode Commands in vi** (on page 3220).
   b. Otherwise, it shall be set to the indentation of the previous current line, if any; otherwise, to the first column.
2. For the *ex* **a**, **i**, and **c** commands, indentation for each line shall be set as follows:
   a. If a line was previously inserted as part of this command, it shall be set to the indentation of the last inserted line by default, or as otherwise specified for the *eof* character in **Scroll** (on page 2567).
   b. Otherwise, if the command is the *ex* **a** command, it shall be set to the line appended after, if any; otherwise to the first column.
   c. Otherwise, if the command is the *ex* **i** command, it shall be set to the line inserted before, if any; otherwise to the first column.
   d. Otherwise, if the command is the *ex* **c** command, it shall be set to the indentation of the line replaced.

autoprint, ap

[Default *set*]

If **autoprint** is set, the current line shall be written after each *ex* command that modifies the contents of the current edit buffer, and after each **tag** command for which the tag search pattern was found or tag line number was valid, unless:

1. The command was executed while in open or visual mode.
2. The command was executed as part of a **global** or **v** command or **@** buffer execution.
3. The command was the form of the **read** command that reads a file into the edit buffer.
4. The command was the **append**, **change**, or **insert** command.
5. The command was not terminated by a <newline>.
6. The current line shall be written by a flag specified to the command; for example, **delete #** shall write the current line as specified for the flag modifier to the **delete** command, and not as specified by the **autoprint** edit option.

autowrite, aw

[Default *unset*]

If **autowrite** is set, and the edit buffer has been modified since it was last completely written to any file, the contents of the edit buffer shall be written as if the *ex* **write** command had been specified without arguments, before each command affected by the **autowrite** edit option is executed. Appending the character '!' to the command name of any of the *ex* commands except '!' shall prevent the write. If the write fails, it shall be an error and the command shall not be executed.

beautify, bf

XSI [Default *unset*]

If **beautify** is set, all non-printable characters, other than <tab>s, <newline>s, and <form-feed>s, shall be discarded from text read in from files.

directory, dir

[Default *implementation-defined*]

The value of this option specifies the directory in which the editor buffer is to be placed. If this directory is not writable by the user, the editor shall quit.

edcompatible, ed

[Default *unset*]

Causes the presence of **g** and **c** suffixes on substitute commands to be remembered, and toggled by repeating the suffixes.

errorbells, eb

[Default *unset*]

If the editor is in *ex* mode, and the terminal does not support a standout mode (such as inverse video), and **errorbells** is set, error messages shall be preceded by alerting the terminal.

exrc

[Default *unset*]

If **exrc** is set, *ex* shall access any **.exrc** file in the current directory, as described in **Initialization in ex and vi** (on page 2559). If **exrc** is not set, *ex* shall ignore any **.exrc** file in the current directory during initialization, unless the current directory is that named by the *HOME* environment variable.

ignorecase, ic

[Default *unset*]

If **ignorecase** is set, characters that have uppercase and lowercase representations shall have those representations considered as equivalent for purposes of regular expression comparison.

The **ignorecase** edit option shall affect all remembered regular expressions; for example, unsetting the **ignorecase** edit option shall cause a subsequent *vi* **n** command to search for the last basic regular expression in a case-sensitive fashion.

list

[Default *unset*]

If **list** is set, edit buffer lines written while in *ex* command mode shall be written as specified for the **print** command with the **l** flag specified. In open or visual mode, each edit buffer line shall be displayed as specified for the *ex* print command with the **l** flag specified. In open or visual text input mode, when the cursor does not rest on any character in the line, it shall rest on the `'$'` marking the end of the line.

magic

[Default *set*]

If **magic** is set, modify the interpretation of characters in regular expressions and substitution replacement strings (see **Regular Expressions in ex** (on page 2592) and **Replacement Strings in ex** (on page 2592)).

mesg

[Default *set*]

If **mesg** is set, the permission for others to use the **write** or **talk** commands to write to the terminal shall be turned on while in open or visual mode. The shell-level command *mesg* **n** shall take precedence over any setting of the *ex* **mesg** option; that is, if **mesg y** was issued before the editor started (or in a shell escape), such as:

```
:!mesg y
```

the **mesg** option in *ex* shall suppress incoming messages, but the **mesg** option shall not enable incoming messages if **mesg n** was issued.

number, nu

[Default *unset*]

If **number** is set, edit buffer lines written while in *ex* command mode shall be written with line numbers, in the format specified by the **print** command with the **#** flag specified. In *ex* text input mode, each line shall be preceded by the line number it will have in the file.

In open or visual mode, each edit buffer line shall be displayed with a preceding line number, in the format specified by the *ex* **print** command with the **#** flag specified. This line number shall not be considered part of the line for the purposes of evaluating the current column; that is, column position 1 shall be the first column position after the format specified by the **print** command.

paragraphs, para

[Default in the POSIX locale `IPLPPPQPP LIpplpipbp`]

The **paragraphs** edit option shall define additional paragraph boundaries for the open and visual mode commands. The **paragraphs** edit option can be set to a character string consisting of zero or more character pairs. It shall be an error to set it to an odd number of characters.

prompt

[Default *set*]

If **prompt** is set, *ex* command mode input shall be prompted for with a colon (`':'`); when unset, no prompt shall be written.

readonly

[Default *see text*]

If **readonly** edit option is set, read-only mode shall be enabled (see **Write** (on page 2587)). The **readonly** edit option shall be initialized to set if either of the following conditions are true:

- The command-line option –R was specified.
- Performing actions equivalent to the *access*( ) function called with the following arguments indicates that the file lacks write permission:
  1. The current pathname is used as the *path* argument.
  2. The constant **W_OK** is used as the *amode* argument.

The **readonly** edit option may be initialized to set for other, implementation-defined reasons. The **readonly** edit option shall not be initialized to unset based on any special privileges of the user or process. The **readonly** edit option shall be reinitialized each time that the contents of the edit buffer are replaced (for example, by an **edit** or **next** command) unless the user has explicitly set it, in which case it shall remain set until the user explicitly unsets it. Once unset, it shall again be reinitialized each time that the contents of the edit buffer are replaced.

redraw

[Default *unset*]

The editor simulates an intelligent terminal on a dumb terminal. (Since this is likely to require a large amount of output to the terminal, it is useful only at high transmission speeds.)

remap

[Default *set*]

If **remap** is set, map translation shall allow for maps defined in terms of other maps; translation shall continue until a final product is obtained. If unset, only a one-step translation shall be done.

report

[Default 5]

The value of this **report** edit option specifies what number of lines being added, copied, deleted, or modified in the edit buffer will cause an informational message to be written to the user. The following conditions shall cause an informational message. The message shall contain the number of lines added, copied, deleted, or modified, but is otherwise unspecified.

- An *ex* or *vi* editor command, other than **open**, **undo**, or **visual**, that modifies at least the value of the **report** edit option number of lines, and which is not part of an *ex* **global** or **v** command, or *ex* or *vi* buffer execution, shall cause an informational message to be written.
- An *ex* **yank** or *vi* **y** or **Y** command, that copies at least the value of the **report** edit option plus 1 number of lines, and which is not part of an *ex* **global** or **v** command, or *ex* or *vi* buffer execution, shall cause an informational message to be written.
- An *ex* **global**, **v**, **open**, **undo**, or **visual** command or *ex* or *vi* buffer execution, that adds or deletes a total of at least the value of the **report** edit option number of lines, and which is not part of an *ex* **global** or **v** command, or *ex* or *vi* buffer execution, shall cause an informational message to be written. (For example, if 3 lines were added and 8 lines deleted during an *ex* **visual** command, 5 would be the number compared against the **report** edit option after the command completed.

scroll, scr

[Default (number of lines in the display −1)/2]

The value of the **scroll** edit option shall determine the number of lines scrolled by the *ex* <control>-D and **z** commands. For the *vi* <control>-D and <control>-U commands, it shall be the initial number of lines to scroll when no previous <control>-D or <control>-U command has been executed.

sections

[Default in the POSIX locale `NHSHH HUnhsh`]

The **sections** edit option shall define additional section boundaries for the open and visual mode commands. The **sections** edit option can be set to a character string consisting of zero or more character pairs; it shall be an error to set it to an odd number of characters.

shell, sh

[Default from the environment variable *SHELL*]

The value of this option shall be a string. The default shall be taken from the *SHELL* environment variable. If the *SHELL* environment variable is null or empty, the *sh* (see *sh*) utility shall be the default.

shiftwidth, sw

[Default 8]

The value of this option shall give the width in columns of an indentation level used during autoindentation and by the shift commands (< and >).

showmatch, sm

[Default *unset*]

The functionality described for the **showmatch** edit option need not be supported on block-mode terminals or terminals with insufficient capabilities.

If **showmatch** is set, in open or visual mode, when a `')'` or `'}'` is typed, if the matching `'('` or `'{'` is currently visible on the display, the matching `'('` or `'{'` shall be flagged moving the cursor to its location for an unspecified amount of time.

showmode

[Default *unset*]

If **showmode** is set, in open or visual mode, the current mode that the editor is in shall be displayed on the last line of the display. Command mode and text input mode shall be differentiated; other unspecified modes and implementation-defined information may be displayed.

slowopen

[Default *unset*]

If **slowopen** is set during open and visual text input modes, the editor shall not update portions of the display other than those display line columns that display the characters entered by the user (see **Input Mode Commands in vi** (on page 3220)).

tabstop, ts

[Default 8]

The value of this edit option shall specify the column boundary used by a <tab> in the display (see **autoprint, ap** (on page 2594) and **Input Mode Commands in vi** (on page 3220)).

taglength, tl

[Default zero]

The value of this edit option shall specify the maximum number of characters that are considered significant in the user-specified tag name and in the tag name from the tags file. If the value is zero, all characters in both tag names shall be significant.

tags

[Default *see text*]

The value of this edit option shall be a string of <blank>-delimited pathnames of files used by the **tag** command. The default value is unspecified.

term

[Default from the environment variable *TERM*]

The value of this edit option shall be a string. The default shall be taken from the *TERM* variable in the environment. If the *TERM* environment variable is empty or null, the default is unspecified. The editor shall use the value of this edit option to determine the type of the display device.

The results are unspecified if the user changes the value of the term edit option after editor initialization.

terse

[Default *unset*]

If **terse** is set, error messages may be less verbose. However, except for this caveat, error messages are unspecified. Furthermore, not all error messages need change for different settings of this option.

warn

[Default *set*]

If **warn** is set, and the contents of the edit buffer have been modified since they were last completely written, the editor shall write a warning message before certain **!** commands (see **Escape** (on page 2590)).

window

[Default *see text*]

A value used in open and visual mode, by the <control>-B and <control>-F commands, and, in visual mode, to specify the number of lines displayed when the screen is repainted.

If the −**w** command-line option is not specified, the default value shall be set to the value of the *LINES* environment variable. If the *LINES* environment variable is empty or null, the default shall be the number of lines in the display minus 1.

Setting the **window** edit option to zero or to a value greater than the number of lines in the display minus 1 (either explicitly or based on the −**w** option or the *LINES* environment variable) shall cause the **window** edit option to be set to the number of lines in the display minus 1.

The baud rate of the terminal line may change the default in an implementation-defined manner.

wrapmargin, wm

[Default 0]

If the value of this edit option is zero, it shall have no effect.

If not in the POSIX locale, the effect of this edit option is implementation-defined.

Otherwise, it shall specify a number of columns from the ending margin of the terminal.

During open and visual text input modes, for each character for which any part of the character is displayed in a column that is less than **wrapmargin** columns from the ending margin of the display line, the editor shall behave as follows:

1. If the character triggering this event is a <blank>, it, and all immediately preceding <blank>s on the current line entered during the execution of the current text input command, shall be discarded, and the editor shall behave as if the user had entered a single <newline> instead. In addition, if the next user-entered character is a <space>, it shall be discarded as well.

2. Otherwise, if there are one or more <blank>s on the current line immediately preceding the last group of inserted non-<blank>s which was entered during the execution of the current text input command, the <blank>s shall be replaced as if the user had entered a single <newline> instead.

If the **autoindent** edit option is set, and the events described in 1. or 2. are performed, any <blank>s at or after the cursor in the current line shall be discarded.

The ending margin shall be determined by the system or overridden by the user, as described for *COLUMNS* in in the ENVIRONMENT VARIABLES section and the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

wrapscan, ws

[Default *set*]

If **wrapscan** is set, searches (the *ex* **/** or **?** addresses, or open and visual mode **/**, **?**, **N**, and **n** commands) shall wrap around the beginning or end of the edit buffer; when unset, searches shall stop at the beginning or end of the edit buffer.

writeany, wa

[Default *unset*]

If **writeany** is set, some of the checks performed when executing the *ex* **write** commands shall be inhibited, as described in editor option **autowrite**.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

When any error is encountered and the standard input is not a terminal device file, *ex* shall not write the file or return to command or text input mode, and shall terminate with a non-zero exit status.

Otherwise, when an unrecoverable error is encountered, it shall be equivalent to a SIGHUP asynchronous event.

Otherwise, when an error is encountered, the editor shall behave as specified in **Command Line Parsing in ex** (on page 2563).

APPLICATION USAGE

If a SIGSEGV signal is received while *ex* is saving a file, the file might not be successfully saved.

The **next** command can accept more than one file, so usage such as:

```
next ‘ls [abc]*‘
```

is valid; it would not be valid for the **edit** or **read** commands, for example, because they expect only one file and unspecified results occur.

EXAMPLES

None.

RATIONALE

The *ex*/*vi* specification is based on the historical practice found in the 4 BSD and System V implementations of *ex* and *vi*. A freely redistributable implementation of *ex*/*vi*, which is tracking IEEE Std 1003.1-200x fairly closely, and demonstrates the intended changes between historical implementations and IEEE Std 1003.1-200x, may be obtained by anonymous FTP from:

```
ftp://ftp.rdg.opengroup/pub/mirrors/nvi
```

A *restricted editor* (both the historical *red* utility and modifications to *ex*) were considered and rejected for inclusion. Neither option provided the level of security that users might expect.

It is recognized that *ex* visual mode and related features would be difficult, if not impossible, to implement satisfactorily on a block-mode terminal, or a terminal without any form of cursor addressing; thus, it is not a mandatory requirement that such features should work on all terminals. It is the intention, however, that an *ex* implementation should provide the full set of capabilities on all terminals capable of supporting them.

Options

The −**c** replacement for +*command* was inspired by the −**e** option of *sed*. Historically, all such commands (see **edit** and **next** as well) were executed from the last line of the edit buffer. This meant, for example, that `"+/pattern"` would fail unless the **wrapscan** option was set. IEEE Std 1003.1-200x requires conformance to historical practice. Historically, some implementations restricted the *ex* commands that could be listed as part of the command line arguments. For consistency, IEEE Std 1003.1-200x does not permit these restrictions.

In historical implementations of the editor, the −**R** option (and the **readonly** edit option) only prevented overwriting of files; appending to files was still permitted, mapping loosely into the *csh* **noclobber** variable. Some implementations, however, have not followed this semantic, and **readonly** does not permit appending either. IEEE Std 1003.1-200x follows the latter practice, believing that it is a more obvious and intuitive meaning of **readonly**.

The −**s** option suppresses all interactive user feedback and is useful for editing scripts in batch jobs. The list of specific effects is historical practice. The terminal type ‘‘incapable of supporting open and visual modes’’ has historically been named ‘‘dumb’’.

The −**t** option was required because the *ctags* utility appears in IEEE Std 1003.1-200x and the option is available in all historical implementations of *ex*.

Historically, the *ex* and *vi* utilities accepted a −**x** option, which did encryption based on the algorithm found in the historical *crypt* utility. The −**x** option for encryption, and the associated *crypt* utility, were omitted because the algorithm used was not specifiable and the export control laws of some nations make it difficult to export cryptographic technology. In addition, it did not historically provide the level of security that users might expect.

Standard Input

An end-of-file condition is not equivalent to an end-of-file character. A common end-of-file character, <control>-D, is historically an *ex* command.

There was no maximum line length in historical implementations of *ex*. Specifically, as it was parsed in chunks, the addresses had a different maximum length than the filenames. Further, the maximum line buffer size was declared as BUFSIZ, which was different lengths on different systems. This version selected the value of {LINE_MAX} to impose a reasonable restriction on portable usage of *ex* and to aid test suite writers in their development of realistic tests that exercise this limit.

Input Files

It was an explicit decision by the standard developers that a <newline> be added to any file lacking one. It was believed that this feature of *ex* and *vi* was relied on by users in order to make text files lacking a trailing <newline> more portable. It is recognized that this will require a user-specified option or extension for implementations that permit *ex* and *vi* to edit files of type other than text if such files are not otherwise identified by the system. It was agreed that the ability to edit files of arbitrary type can be useful, but it was not considered necessary to mandate that an *ex* or *vi* implementation be required to handle files other than text files.

The paragraph in the INPUT FILES section, ‘‘By default, . . .’’, is intended to close a long-standing security problem in *ex* and *vi*, that of the ‘‘modeline’’ or ‘‘modelines’’ edit option. This feature allows any line in the first or last five lines of the file containing the strings `"ex:"` or `"vi:"` (and, apparently, `"ei:"` or `"vx:"`) to be a line containing editor commands, and *ex* interprets all the text up to the next `':'` or <newline> as a command. Consider the consequences, for example, of an unsuspecting user using *ex* or *vi* as the editor when replying to a mail message in which a line such as:

```
ex:! rm −rf :
```

appeared in the signature lines. The standard developers believed strongly that an editor should not by default interpret any lines of a file. Vendors are strongly urged to delete this feature from their implementations of *ex* and *vi*.

Asynchronous Events

The intention of the phrase ‘‘complete write’’ is that the entire edit buffer be written to stable storage. The note regarding temporary files is intended for implementations that use temporary files to back edit buffers unnamed by the user.

Historically, SIGQUIT was ignored by *ex*, but was the equivalent of the **Q** command in visual mode; that is, it exited visual mode and entered *ex* mode. IEEE Std 1003.1-200x permits, but does not require, this behavior. Historically, SIGINT was often used by *vi* users to terminate text input mode (<control>-C is often easier to enter than <ESC>). Some implementations of *vi* alerted the terminal on this event, and some did not. IEEE Std 1003.1-200x requires that SIGINT behave identically to <ESC>, and that the terminal not be alerted.

Historically, suspending the *ex* editor during text input mode was similar to SIGINT, as completed lines were retained, but any partial line discarded, and the editor returned to command mode. IEEE Std 1003.1-200x is silent on this issue; implementations are encouraged to follow historical practice, where possible.

Historically, the *vi* editor did not treat SIGTSTP as an asynchronous event, and it was therefore impossible to suspend the editor in visual text input mode. There are two major reasons for this. The first is that SIGTSTP is a broadcast signal on UNIX systems, and the chain of events where the shell *exec*s an application that then *exec*s *vi* usually caused confusion for the terminal state if SIGTSTP was delivered to the process group in the default manner. The second was that most implementations of the UNIX *curses* package are not reentrant, and the receipt of SIGTSTP at the wrong time will cause them to crash. IEEE Std 1003.1-200x is silent on this issue; implementations are encouraged to treat suspension as an asynchronous event if possible.

Historically, modifications to the edit buffer made before SIGINT interrupted an operation were retained; that is, anywhere from zero to all of the lines to be modified might have been modified by the time the SIGINT arrived. These changes were not discarded by the arrival of SIGINT. IEEE Std 1003.1-200x permits this behavior, noting that the *undo* command is required to be able to undo these partially completed commands.

The action taken for signals other than SIGINT, SIGCONT, SIGHUP, and SIGTERM is unspecified because some implementations attempt to save the edit buffer in a useful state when other signals are received.

Standard Error

For *ex*/*vi*, diagnostic messages are those messages reported as a result of a failed attempt to invoke *ex* or *vi*, such as invalid options or insufficient resources, or an abnormal termination condition. Diagnostic messages should not be confused with the error messages generated by inappropriate or illegal user commands.

Initialization in ex and vi

If an *ex* command (other than **cd**, **chdir**, or **source**) has a filename argument, one or both of the alternate and current pathnames will be set. Informally, they are set as follows:

1. If the *ex* command is one that replaces the contents of the edit buffer, and it succeeds, the current pathname will be set to the filename argument (the first filename argument in the case of the **next** command) and the alternate pathname will be set to the previous current pathname, if there was one.
2. In the case of the file read/write forms of the **read** and **write** commands, if there is no current pathname, the current pathname will be set to the filename argument.
3. Otherwise, the alternate pathname will be set to the filename argument.

For example, **:edit foo** and **:recover foo**, when successful, set the current pathname, and, if there was a previous current pathname, the alternate pathname. The commands **:write**, **!command**, and **:edit** set neither the current or alternate pathnames. If the **:edit foo** command were to fail for some reason, the alternate pathname would be set. The **read** and **write** commands set the alternate pathname to their *file* argument, unless the current pathname is not set, in which case they set the current pathname to their *file* arguments. The alternate pathname was not historically set by the **:source** command. IEEE Std 1003.1-200x requires conformance to historical practice. Implementations adding commands that take filenames as arguments are encouraged to set the alternate pathname as described here.

Historically, *ex* and *vi* read the **.exrc** file in the *$HOME* directory twice, if the editor was executed in the *$HOME* directory. IEEE Std 1003.1-200x prohibits this behavior.

Historically, the 4 BSD *ex* and *vi* read the *$HOME* and local **.exrc** files if they were owned by the real ID of the user, or the **sourceany** option was set, regardless of other considerations. This was a security problem because it is possible to put normal UNIX system commands inside a **.exrc** file. IEEE Std 1003.1-200x does not specify the **sourceany** option, and historical implementations are encouraged to delete it.

The **.exrc** files must be owned by the real ID of the user, and not writeable by anyone other than the owner. The appropriate privileges exception is intended to permit users to acquire special privileges, but continue to use the **.exrc** files in their home directories.

System V Release 3.2 and later *vi* implementations added the option **[no]exrc**. The behavior is that local **.exrc** files are read-only if the **exrc** option is set. The default for the **exrc** option was off, so by default, local **.exrc** files were not read. The problem this was intended to solve was that System V permitted users to give away files, so there is no possible ownership or writeability test to ensure that the file is safe. This is still a security problem on systems where users can give away files, but there is nothing additional that IEEE Std 1003.1-200x can do. The implementation-defined exception is intended to permit groups to have local **.exrc** files that are shared by users, by creating pseudo-users to own the shared files.

IEEE Std 1003.1-200x does not mention system-wide *ex* and *vi* start-up files. While they exist in several implementations of *ex* and *vi*, they are not present in any implementations considered historical practice by IEEE Std 1003.1-200x. Implementations that have such files should use them only if they are owned by the real user ID or an appropriate user (for example, root on UNIX systems) and if they are not writeable by any user other than their owner. System-wide start-up files should be read before the *EXINIT* variable, **$HOME/.exrc** or local **.exrc** files are evaluated.

Historically, any *ex* command could be entered in the *EXINIT* variable or the **.exrc** file, although ones requiring that the edit buffer already contain lines of text generally caused historical implementations of the editor to drop core. IEEE Std 1003.1-200x requires that any *ex* command be permitted in the *EXINIT* variable and **.exrc** files, for simplicity of specification and consistency, although many of them will obviously fail under many circumstances.

The initialization of the contents of the edit buffer uses the phrase ‘‘the effect shall be’’ with regard to various *ex* commands. The intent of this phrase is that edit buffer contents loaded during the initialization phase not be lost; that is, loading the edit buffer should fail if the **.exrc** file read in the contents of a file and did not subsequently write the edit buffer. An additional intent of this phrase is to specify that the initial current line and column is set as specified for the individual *ex* commands.

Historically, the −**t** option behaved as if the tag search were a +*command*; that is, it was executed from the last line of the file specified by the tag. This resulted in the search failing if the pattern was a forward search pattern and the **wrapscan** edit option was not set. IEEE Std 1003.1-200x does not permit this behavior, requiring that the search for the tag pattern be performed on the entire file, and, if not found, that the current line be set to a more reasonable location in the file.

Historically, the empty edit buffer presented for editing when a file was not specified by the user was unnamed. This is permitted by IEEE Std 1003.1-200x; however, implementations are encouraged to provide users a temporary filename for this buffer because it permits them the use of *ex* commands that use the current pathname during temporary edit sessions.

Historically, the file specified using the −**t** option was not part of the current argument list. This practice is permitted by IEEE Std 1003.1-200x; however, implementations are encouraged to include its name in the current argument list for consistency.

Historically, the −**c** command was generally not executed until a file that already exists was edited. IEEE Std 1003.1-200x requires conformance to this historical practice. Commands that could cause the −**c** command to be executed include the *ex* commands **edit**, **next**, **recover**, **rewind**, and **tag**, and the *vi* commands <control>-ˆ and <control>-]. Historically, reading a file into an edit buffer did not cause the −**c** command to be executed (even though it might set the current pathname) with the exception that it did cause the **−c** command to be executed if: the editor was in *ex* mode, the edit buffer had no current pathname, the edit buffer was empty, and no read commands had yet been attempted. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Historically, the **−r** option was the same as a normal edit session if there was no recovery information available for the file. This allowed users to enter:

```
vi −r *.c
```

and recover whatever files were recoverable. In some implementations, recovery was attempted only on the first file named, and the file was not entered into the argument list; in others, recovery was attempted for each file named. In addition, some historical implementations ignored **−r** if **−t** was specified or did not support command line *file* arguments with the **−t** option. For consistency and simplicity of specification, IEEE Std 1003.1-200x disallows these special cases, and requires that recovery be attempted the first time each file is edited.

Historically, *vi* initialized the **‘** and **’** marks, but *ex* did not. This meant that if the first command in *ex* mode was **visual** or if an *ex* command was executed first (for example, *vi* +10 *file*), *vi* was entered without the marks being initialized. Because the standard developers believed the marks to be generally useful, and for consistency and simplicity of specification, IEEE Std 1003.1-200x requires that they always be initialized if in open or visual mode, or if in *ex* mode and the edit buffer is not empty. Not initializing it in *ex* mode if the edit buffer is empty is historical practice; however, it has always been possible to set (and use) marks in empty edit buffers in open and visual mode edit sessions.

Addressing

Historically, *ex* and *vi* accepted the additional addressing forms `'\/'` and `'\?'`. They were equivalent to `"//"` and `"??"`, respectively. They are not required by IEEE Std 1003.1-200x, mostly because nobody can remember whether they ever did anything different historically.

Historically, *ex* and *vi* permitted an address of zero for several commands, and permitted the **%** address in empty files for others. For consistency, IEEE Std 1003.1-200x requires support for the former in the few commands where it makes sense, and disallows it otherwise. In addition, because IEEE Std 1003.1-200x requires that **%** be logically equivalent to `"1,$"`, it is also supported where it makes sense and disallowed otherwise.

Historically, the **%** address could not be followed by further addresses. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that additional addresses be supported.

All of the following are valid *addresses*: |

`+++`	Three lines after the current line.
`/re/−`	One line before the next occurrence of *re*.
`−2`	Two lines before the current line.
`3 −−−− 2`	Line one (note intermediate negative address).
`1 2 3`	Line six.

Any number of addresses can be provided to commands taking addresses; for example, `"1,2,3,4,5p"` prints lines 4 and 5, because two is the greatest valid number of addresses accepted by the **print** command. This, in combination with the semicolon delimiter, permits users to create commands based on ordered patterns in the file. For example, the command **3;/foo/;+2print** will display the first line after line 3 that contains the pattern *foo*, plus the next two lines. Note that the address **3;** must be evaluated before being discarded because the search origin for the **/foo/** command depends on this.

Historically, values could be added to addresses by including them after one or more <blank>s; for example, **3 − 5p** wrote the seventh line of the file, and **/foo/ 5** was the same as **/foo/+5**. However, only absolute values could be added; for example, **5 /foo/** was an error. IEEE Std 1003.1-200x requires conformance to historical practice. Address offsets are separately specified from addresses because they could historically be provided to visual mode search commands.

Historically, any missing addresses defaulted to the current line. This was true for leading and trailing comma-delimited addresses, and for trailing semicolon-delimited addresses. For consistency, IEEE Std 1003.1-200x requires it for leading semicolon addresses as well.

Historically, *ex* and *vi* accepted the `'^'` character as both an address and as a flag offset for commands. In both cases it was identical to the `'−'` character. IEEE Std 1003.1-200x does not require or prohibit this behavior.

Historically, the enhancements to basic regular expressions could be used in addressing; for example, `'~'`, `'\<'`, and `'\>'`. IEEE Std 1003.1-200x requires conformance to historical practice; that is, that regular expression usage be consistent, and that regular expression enhancements be supported wherever regular expressions are used.

Command Line Parsing in ex

Historical *ex* command parsing was even more complex than that described here. IEEE Std 1003.1-200x requires the subset of the command parsing that the standard developers believed was documented and that users could reasonably be expected to use in a portable fashion, and that was historically consistent between implementations. (The discarded functionality is obscure, at best.) Historical implementations will require changes in order to comply with IEEE Std 1003.1-200x; however, users are not expected to notice any of these changes. Most of the complexity in *ex* parsing is to handle three special termination cases:

1. The **!**, **global**, **v**, and the filter versions of the **read** and **write** commands are delimited by <newline>s (they can contain vertical-line characters that are usually shell pipes).
2. The **ex**, **edit**, **next**, and **visual** in open and visual mode commands all take *ex* commands, optionally containing vertical-line characters, as their first arguments.
3. The **s** command takes a regular expression as its first argument, and uses the delimiting characters to delimit the command.

Historically, vertical-line characters in the **+***command* argument of the **ex**, **edit**, **next**, **vi**, and **visual** commands, and in the *pattern* and *replacement* parts of the **s** command, did not delimit the command, and in the filter cases for **read** and **write**, and the **!**, **global**, and **v** commands, they did not delimit the command at all. For example, the following commands are all valid:

```
:edit +25 | s/abc/ABC/ file.c
:s/ | /PIPE/
:read !spell % | columnate
:global/pattern/p | l
:s/a/b/ | s/c/d | set
```

Historically, empty or <blank> filled lines in **.exrc** files and **source**d files (as well as *EXINIT* variables and *ex* command scripts) were treated as default commands; that is, **print** commands. IEEE Std 1003.1-200x specifically requires that they be ignored when encountered in **.exrc** and **source**d files to eliminate a common source of new user error.

Historically, *ex* commands with multiple adjacent (or <blank>-separated) vertical lines were handled oddly when executed from *ex* mode. For example, the command **| | |** <carriage-return>, when the cursor was on line 1, displayed lines 2, 3, and 5 of the file. In addition, the command **|** would only display the line after the next line, instead of the next two lines. The former worked more logically when executed from *vi* mode, and displayed lines 2, 3, and 4. IEEE Std 1003.1-200x requires the *vi* behavior; that is, a single default command and line number increment for each command separator, and trailing <newline>s after vertical-line separators are discarded.

Historically, *ex* permitted a single extra colon as a leading command character; for example, **:g/pattern/:p** was a valid command. IEEE Std 1003.1-200x generalizes this to require that any number of leading colon characters be stripped.

Historically, any prefix of the **delete** command could be followed without intervening <blank>s by a flag character because in the command **d p**, *p* is interpreted as the buffer *p*. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the **k** command could be followed by the mark name without intervening <blank>s. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the **s** command could be immediately followed by flag and option characters; for example, **s/e/E/|s|sgc3p** was a valid command. However, flag characters could not stand alone; for example, the commands **sp** and **s l** would fail, while the command **sgp** and **s gl** would succeed. (Obviously, the '#' flag character was used as a delimiter character if it followed the command.) Another issue was that option characters had to precede flag characters even when the command was fully specified; for example, the command **s/e/E/pg** would fail, while the command **s/e/E/gp** would succeed. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the first command name that had a prefix matching the input from the user was the executed command; for example, **ve**, **ver**, and **vers** all executed the **version** command. Commands were in a specific order, however, so that **a** matched **append**, not **abbreviate**. IEEE Std 1003.1-200x requires conformance to historical practice. The restriction on command search order for implementations with extensions is to avoid the addition of commands such that the historical prefixes would fail to work portably.

Historical implementations of *ex* and *vi* did not correctly handle multiple *ex* commands, separated by vertical-line characters, that entered or exited visual mode or the editor. Because implementations of *vi* exist that do not exhibit this failure mode, IEEE Std 1003.1-200x does not permit it.

The requirement that alphabetic command names consist of all following alphabetic characters up to the next non-alphabetic character means that alphabetic command names must be separated from their arguments by one or more non-alphabetic characters, normally a <blank> or '!' character, except as specified for the exceptions, the **delete**, **k**, and **s** commands.

Historically, the repeated execution of the *ex* default **print** commands (<control>-D, *eof*, <newline>, <carriage-return>) erased any prompting character and displayed the next lines without scrolling the terminal; that is, immediately below any previously displayed lines. This provided a cleaner presentation of the lines in the file for the user. IEEE Std 1003.1-200x does not require this behavior because it may be impossible in some situations; however, implementations are strongly encouraged to provide this semantic if possible.

Historically, it was possible to change files in the middle of a command, and have the rest of the command executed in the new file; for example:

```
:edit +25 file.c | s/abc/ABC/ | 1
```

was a valid command, and the substitution was attempted in the newly edited file. IEEE Std 1003.1-200x requires conformance to historical practice. The following commands are examples that exercise the *ex* parser:

```
echo 'foo | bar' > file1; echo 'foo/bar' > file2;
vi
:edit +1 | s/|/PIPE/ | w file1 | e file2 | 1 | s/\//SLASH/ | wq
```

Historically, there was no protection in editor implementations to avoid *ex* **global**, **v**, **@**, or ***** commands changing edit buffers during execution of their associated commands. Because this would almost invariably result in catastrophic failure of the editor, and implementations exist that do exhibit these problems, IEEE Std 1003.1-200x requires that changing the edit buffer during a **global** or **v** command, or during a **@** or ***** command for which there will be more than a single execution, be an error. Implementations supporting multiple edit buffers simultaneously are strongly encouraged to apply the same semantics to switching between buffers as well.

The *ex* command quoting required by IEEE Std 1003.1-200x is a superset of the quoting in historical implementations of the editor. For example, it was not historically possible to escape a <blank> in a filename; for example, **:edit foo**`\\\` **bar** would report that too many filenames had been entered for the edit command, and there was no method of escaping a <blank> in the first argument of an **edit**, **ex**, **next**, or **visual** command at all. IEEE Std 1003.1-200x extends historical practice, requiring that quoting behavior be made consistent across all *ex* commands, except for the **map**, **unmap**, **abbreviate**, and **unabbreviate** commands, which historically used <control>-V instead of backslashes for quoting. For those four commands, IEEE Std 1003.1-200x requires conformance to historical practice.

Backslash quoting in *ex* is non-intuitive. Backslash escapes are ignored unless they escape a special character; for example, when performing *file* argument expansion, the string `"\\%"` is equivalent to `'\%'`, not `"\<`*current path name*`>"`. This can be confusing for users because backslash is usually one of the characters that causes shell expansion to be performed, and therefore shell quoting rules must be taken into consideration. Generally, quoting characters are only considered if they escape a special character, and a quoting character must be provided for each layer of parsing for which the character is special. As another example, only a single backslash is necessary for the `'\l'` sequence in substitute replacement patterns, because the character `'l'` is not special to any parsing layer above it.

<control>-V quoting in *ex* is slightly different from backslash quoting. In the four commands where <control>-V quoting applies (**abbreviate**, **unabbreviate**, **map**, and **unmap**), any character may be escaped by a <control>-V whether it would have a special meaning or not. IEEE Std 1003.1-200x requires conformance to historical practice.

Historical implementations of the editor did not require delimiters within character classes to be escaped; for example, the command **:s/[/]//** on the string `"xxx/yyy"` would delete the `'/'` from the string. IEEE Std 1003.1-200x disallows this historical practice for consistency and because it places a large burden on implementations by requiring that knowledge of regular expressions be built into the editor parser.

Historically, quoting <newline>s in *ex* commands was handled inconsistently. In most cases, the <newline> always terminated the command, regardless of any preceding escape character, because backslash characters did not escape <newline>s for most *ex* commands. However, some *ex* commands (for example, **s**, **map**, and **abbreviation**) permitted <newline>s to be escaped (although in the case of **map** and **abbreviation**, <control>-V characters escaped them instead of backslashes). This was true in not only the command line, but also **.exrc** and **source**d files. For example, the command:

```
map = foo<control-V><newline>bar
```

would succeed, although it was sometimes difficult to get the <control>-V and the inserted <newline> passed to the *ex* parser. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that it be possible to escape <newline>s in *ex* commands at all times, using backslashes for most *ex* commands, and using <control>-V characters for the **map** and **abbreviation** commands. For example, the command **print**<newline>**list** is required to be parsed as the single command **print**<newline>**list**. While this differs from historical practice, IEEE Std 1003.1-200x developers believed it unlikely that any script or user depended on the historical behavior.

Historically, an error in a command specified using the **–c** option did not cause the rest of the **–c** commands to be discarded. IEEE Std 1003.1-200x disallows this for consistency with mapped keys, the **@**, **global**, **source**, and **v** commands, the *EXINIT* environment variable, and the **.exrc** files.

Input Editing in ex

One of the common uses of the historical *ex* editor is over slow network connections. Editors that run in canonical mode can require far less traffic to and from, and far less processing on, the host machine, as well as more easily supporting block-mode terminals. For these reasons, IEEE Std 1003.1-200x requires that *ex* be implemented using canonical mode input processing, as was done historically.

IEEE Std 1003.1-200x does not require the historical 4 BSD input editing characters ‘‘word erase’’ or ‘‘literal next’’. For this reason, it is unspecified how they are handled by *ex*, although they must have the required effect. Implementations that resolve them after the line has been ended using a <newline> or <control>-M character, and implementations that rely on the underlying system terminal support for this processing, are both conforming. Implementations are strongly urged to use the underlying system functionality, if at all possible, for compatibility with other system text input interfaces.

Historically, when the *eof* character was used to decrement the **autoindent** level, the cursor moved to display the new end of the **autoindent** characters, but did not move the cursor to a new line, nor did it erase the <control>-D character from the line. IEEE Std 1003.1-200x does not specify that the cursor remain on the same line or that the rest of the line is erased; however, implementations are strongly encouraged to provide the best possible user interface; that is, the cursor should remain on the same line, and any <control>-D character on the line should be erased.

IEEE Std 1003.1-200x does not require the historical 4 BSD input editing character ‘‘reprint’’, traditionally <control>-R, which redisplayed the current input from the user. For this reason, and because the functionality cannot be implemented after the line has been terminated by the user, IEEE Std 1003.1-200x makes no requirements about this functionality. Implementations are strongly urged to make this historical functionality available, if possible.

Historically, <control>-Q did not perform a literal next function in *ex*, as it did in *vi*. IEEE Std 1003.1-200x requires conformance to historical practice to avoid breaking historical *ex* scripts and **.exrc** files.

eof

Whether the *eof* character immediately modifies the **autoindent** characters in the prompt is left unspecified so that implementations can conform in the presence of systems that do not support this functionality. Implementations are encouraged to modify the line and redisplay it immediately, if possible.

The specification of the handling of the *eof* character differs from historical practice only in that *eof* characters are not discarded if they follow normal characters in the text input. Historically, they were always discarded.

Command Descriptions in ex

Historically, several commands (for example, **global**, **v**, **visual**, **s**, **write**, **wq**, **yank**, **!**, **<**, **>**, **&**, and **~**) were executable in empty files (that is, the default address(es) were 0), or permitted explicit addresses of 0 (for example, 0 was a valid address, or 0,0 was a valid range). Addresses of 0, or command execution in an empty file, make sense only for commands that add new text to the edit buffer or write commands (because users may wish to write empty files). IEEE Std 1003.1-200x requires this behavior for such commands and disallows it otherwise, for consistency and simplicity of specification.

A count to an *ex* command has been historically corrected to be no greater than the last line in a file; for example, in a five-line file, the command **1,6print** would fail, but the command **1print300** would succeed. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the use of flags in *ex* commands could be obscure. General historical practice was as described by IEEE Std 1003.1-200x, but there were some special cases. For example, the **list**, **number**, and **print** commands ignored trailing address offsets; for example, **3p +++#** would display line 3, and 3 would be the current line after the execution of the command. The **open** and **visual** commands ignored both the trailing offsets and the trailing flags. Also, flags specified to the **open** and **visual** commands interacted badly with the **list** edit option, and setting and then unsetting it during the open/visual session would cause *vi* to stop displaying lines in the specified format. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit any of these exceptions to the general rule.

IEEE Std 1003.1-200x uses the word *copy* in several places when discussing buffers. This is not intended to imply implementation.

Historically, *ex* users could not specify numeric buffers because of the ambiguity this would cause; for example, in the command **3 delete 2**, it is unclear whether 2 is a buffer name or a *count*. IEEE Std 1003.1-200x requires conformance to historical practice by default, but does not preclude extensions.

Historically, the contents of the unnamed buffer were frequently discarded after commands that did not explicitly affect it; for example, when using the **edit** command to switch files. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

The *ex* utility did not historically have access to the numeric buffers, and, furthermore, deleting lines in *ex* did not modify their contents. For example, if, after doing a delete in *vi*, the user switched to *ex*, did another delete, and then switched back to *vi*, the contents of the numeric buffers would not have changed. IEEE Std 1003.1-200x requires conformance to historical practice. Numeric buffers are described in the *ex* utility in order to confine the description of buffers to a single location in IEEE Std 1003.1-200x.

The metacharacters that trigger shell expansion in *file* arguments match historical practice, as does the method for doing shell expansion. Implementations wishing to provide users with the flexibility to alter the set of metacharacters are encouraged to provide a **shellmeta** string edit option.

Historically, *ex* commands executed from *vi* refreshed the screen when it did not strictly need to do so; for example, **:!date > /dev/null** does not require a screen refresh because the output of the UNIX *date* command requires only a single line of the screen. IEEE Std 1003.1-200x requires that the screen be refreshed if it has been overwritten, but makes no requirements as to how an implementation should make that determination. Implementations may prompt and refresh the screen regardless.

Abbreviate

Historical practice was that characters that were entered as part of an abbreviation replacement were subject to **map** expansions, the **showmatch** edit option, further abbreviation expansions, and so on; that is, they were logically pushed onto the terminal input queue, and were not a simple replacement. IEEE Std 1003.1-200x requires conformance to historical practice. Historical practice was that whenever a non-word character (that had not been escaped by a <control>-V) was entered after a word character, *vi* would check for abbreviations. The check was based on the type of the character entered before the word character of the word/non-word pair that triggered the check. The word character of the word/non-word pair that triggered the check and all characters entered before the trigger pair that were of that type were included in the check, with the exception of <blank>s, which always delimited the abbreviation.

This means that, for the abbreviation to work, the *lhs* must end with a word character, there can be no transitions from word to non-word characters (or *vice versar*) other than between the last and next-to-last characters in the *lhs*, and there can be no <blank>s in the *lhs*. In addition, because of the historical quoting rules, it was impossible to enter a literal <control>-V in the *lhs*. IEEE Std 1003.1-200x requires conformance to historical practice. Historical implementations did not inform users when abbreviations that could never be used were entered; implementations are strongly encouraged to do so.

For example, the following abbreviations will work:

```
:ab (p  REPLACE
:ab p   REPLACE
:ab ((p REPLACE
```

The following abbreviations will not work:

```
:ab (   REPLACE
:ab (pp REPLACE
```

Historical practice is that words on the *vi* colon command line were subject to abbreviation expansion, including the arguments to the **abbrev** (and more interestingly) the **unabbrev** command. Because there are implementations that do not do abbreviation expansion for the first argument to those commands, this is permitted, but not required, by IEEE Std 1003.1-200x. However, the following sequence:

```
:ab foo bar
:ab foo baz
```

resulted in the addition of an abbreviation of `"baz"` for the string `"bar"` in historical *ex*/*vi*, and the sequence:

```
:ab foo1 bar
:ab foo2 bar
:unabbreviate foo2
```

deleted the abbreviation `"foo1"`, not `"foo2"`. These behaviors are not permitted by IEEE Std 1003.1-200x because they clearly violate the expectations of the user.

It was historical practice that <control>-V, not backslash, characters be interpreted as escaping subsequent characters in the **abbreviate** command. IEEE Std 1003.1-200x requires conformance to historical practice; however, it should be noted that an abbreviation containing a <blank> will never work.

Append

Historically, any text following a vertical-line command separator after an **append**, **change**, or **insert** command became part of the insert text. For example, in the command:

```
:g/pattern/append|stuff1
```

a line containing the text `"stuff1"` would be appended to each line matching pattern. It was also historically valid to enter:

```
:append|stuff1
stuff2
.
```

and the text on the *ex* command line would be appended along with the text inserted after it. There was an historical bug, however, that the user had to enter two terminating lines (the `'.'` lines) to terminate text input mode in this case. IEEE Std 1003.1-200x requires conformance to historical practice, but disallows the historical need for multiple terminating lines.

Change

See the RATIONALE for the **append** command. Historical practice for cursor positioning after the change command when no text is input, is as described in IEEE Std 1003.1-200x. However, one System V implementation is known to have been modified such that the cursor is positioned on the first address specified, and not on the line before the first address. IEEE Std 1003.1-200x disallows this modification for consistency.

Historically, the **change** command did not support buffer arguments, although some implementations allow the specification of an optional buffer. This behavior is neither required nor disallowed by IEEE Std 1003.1-200x.

Change Directory

A common extension in *ex* implementations is to use the elements of a **cdpath** edit option as prefix directories for *path* arguments to **chdir** that are relative pathnames and that do not have `'.'` or `".."` as their first component. Elements in the **cdpath** edit option are colon-separated. The initial value of the **cdpath** edit option is the value of the shell *CDPATH* environment variable. This feature was not included in IEEE Std 1003.1-200x because it does not exist in any of the implementations considered historical practice.

Copy

Historical implementations of *ex* permitted copies to lines inside of the specified range; for example, **:2,5copy3** was a valid command. IEEE Std 1003.1-200x requires conformance to historical practice.

Delete

IEEE Std 1003.1-200x requires support for the historical parsing of a **delete** command followed by flags, without any intervening <blank>s. For example:

1dp	Deletes the first line and prints the line that was second.
1delep	As for **1dp**.
1d	Deletes the first line, saving it in buffer *p*.
1d p1l	(Pee-one-ell.) Deletes the first line, saving it in buffer *p*, and listing the line that was second.

Edit

Historically, any *ex* command could be entered as a **+***command* argument to the **edit** command, although some (for example, **insert** and **append**) were known to confuse historical implementations. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that any command be supported as an argument to the **edit** command.

Historically, the command argument was executed with the current line set to the last line of the file, regardless of whether the **edit** command was executed from visual mode or not. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the **+***command* specified to the **edit** and **next** commands was delimited by the first <blank>, and there was no way to quote them. For consistency, IEEE Std 1003.1-200x requires that the usual *ex* backslash quoting be provided.

Historically, specifying the **+***command* argument to the edit command required a filename to be specified as well; for example, **:edit +100** would always fail. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this usage to fail for that reason.

Historically, only the cursor position of the last file edited was remembered by the editor. IEEE Std 1003.1-200x requires that this be supported; however, implementations are permitted to remember and restore the cursor position for any file previously edited.

File

Historical versions of the *ex* editor **file** command displayed a current line and number of lines in the edit buffer of 0 when the file was empty, while the *vi* <control>-G command displayed a current line and number of lines in the edit buffer of 1 in the same situation. IEEE Std 1003.1-200x does not permit this discrepancy, instead requiring that a message be displayed indicating that the file is empty.

Global

The two-pass operation of the **global** and **v** commands is not intended to imply implementation, only the required result of the operation.

The current line and column are set as specified for the individual *ex* commands. This requirement is cumulative; that is, the current line and column must track across all the commands executed by the **global** or **v** commands.

Insert

See the RATIONALE for the **append** command.

Historically, **insert** could not be used with an address of zero; that is, not when the edit buffer was empty. IEEE Std 1003.1-200x requires that this command behave consistently with the **append** command.

Join

The action of the **join** command in relation to the special characters is only defined for the POSIX locale because the correct amount of white space after a period varies; in Japanese none is required, in French only a single space, and so on.

List

The historical output of the **list** command was potentially ambiguous. The standard developers believed correcting this to be more important than adhering to historical practice, and IEEE Std 1003.1-200x requires unambiguous output.

Map

Historically, command mode maps only applied to command names; for example, if the character `'x'` was mapped to `'y'`, the command **fx** searched for the `'x'` character, not the `'y'` character. IEEE Std 1003.1-200x requires this behavior. Historically, entering <control>-V as the first character of a *vi* command was an error. Several implementations have extended the semantics of *vi* such that <control>-V means that the subsequent command character is not mapped. This is permitted, but not required, by IEEE Std 1003.1-200x. Regardless, using <control>-V to escape the second or later character in a sequence of characters that might match a **map** command, or any character in text input mode, is historical practice, and stops the entered keys from matching a map. IEEE Std 1003.1-200x requires conformance to historical practice.

Historical implementations permitted digits to be used as a **map** command *lhs*, but then ignored the map. IEEE Std 1003.1-200x requires that the mapped digits not be ignored.

The historical implementation of the **map** command did not permit **map** commands that were more than a single character in length if the first character was printable. This behavior is permitted, but not required, by IEEE Std 1003.1-200x.

Historically, mapped characters were remapped unless the **remap** edit option was not set, or the prefix of the mapped characters matched the mapping characters; for example, in the **map**:

```
:map ab abcd
```

the characters `"ab"` were used as is and were not remapped, but the characters `"cd"` were mapped if appropriate. This can cause infinite loops in the *vi* mapping mechanisms. IEEE Std 1003.1-200x requires conformance to historical practice, and that such loops be interruptible.

Text input maps had the same problems with expanding the *lhs* for the *ex* **map!** and **unmap!** command as did the *ex* **abbreviate** and **unabbreviate** commands. See the RATIONALE for the *ex* **abbreviate** command. IEEE Std 1003.1-200x requires similar modification of some historical practice for the **map** and **unmap** commands, as described for the **abbreviate** and **unabbreviate** commands.

Historically, **map**s that were subsets of other **map**s behaved differently depending on the order in which they were defined. For example:

```
:map! ab     short
:map! abc    long
```

would always translate the characters `"ab"` to `"short"`, regardless of how fast the characters `"abc"` were entered. If the entry order was reversed:

```
:map! abc    long
:map! ab     short
```

the characters `"ab"` would cause the editor to pause, waiting for the completing `'c'` character, and the characters might never be mapped to `"short"`. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that the shortest match be used at all times.

The length of time the editor spends waiting for the characters to complete the *lhs* is unspecified because the timing capabilities of systems are often inexact and variable, and it may depend on other factors such as the speed of the connection. The time should be long enough for the user to be able to complete the sequence, but not long enough for the user to have to wait. Some implementations of *vi* have added a **keytime** option, which permits users to set the number of 0,1 seconds the editor waits for the completing characters. Because mapped terminal function and cursor keys tend to start with an <ESC> character, and <ESC> is the key ending *vi* text input mode, **map**s starting with <ESC> characters are generally exempted from this timeout period, or, at least timed out differently.

Mark

Historically, users were able to set the ‘‘previous context’’ marks explicitly. In addition, the *ex* commands **’** and **'** and the *vi* commands **’**, **‘**, **'**, and **'** all referred to the same mark. In addition, the previous context marks were not set if the command, with which the address setting the mark was associated, failed. IEEE Std 1003.1-200x requires conformance to historical practice. Historically, if marked lines were deleted, the mark was also deleted, but would reappear if the change was undone. IEEE Std 1003.1-200x requires conformance to historical practice.

The description of the special events that set the **‘** and **’** marks matches historical practice. For example, historically the command **/a/,/b/** did not set the **‘** and **’** marks, but the command **/a/,/b/delete** did.

Next

Historically, any *ex* command could be entered as a +*command* argument to the **next** command, although some (for example, **insert** and **append**) were known to confuse historical implementations. IEEE Std 1003.1-200x requires that any command be permitted and that it behave as specified. The **next** command can accept more than one file, so usage such as:

```
next ‘ls [abc] ‘
```

is valid; it need not be valid for the **edit** or **read** commands, for example, because they expect only one filename.

Historically, the **next** command behaved differently from the **:rewind** command in that it ignored the force flag if the **autowrite** flag was set. For consistency, IEEE Std 1003.1-200x does not permit this behavior.

Historically, the **next** command positioned the cursor as if the file had never been edited before, regardless. IEEE Std 1003.1-200x does not permit this behavior, for consistency with the **edit** command.

Implementations wanting to provide a counterpart to the **next** command that edited the previous file have used the command **prev[ious],** which takes no *file* argument.

IEEE Std 1003.1-200x does not require this command.

Open

Historically, the **open** command would fail if the **open** edit option was not set. IEEE Std 1003.1-200x does not mention the **open** edit option and does not require this behavior. Some historical implementations do not permit entering open mode from open or visual mode, only from *ex* mode. For consistency, IEEE Std 1003.1-200x does not permit this behavior.

Historically, entering open mode from the command line (that is, *vi* **+open**) resulted in anomalous behaviors; for example, the *ex* file and *set* commands, and the *vi* command <control>-G did not work. For consistency, IEEE Std 1003.1-200x does not permit this behavior.

Historically, the **open** command only permitted `'/'` characters to be used as the search pattern delimiter. For consistency, IEEE Std 1003.1-200x requires that the search delimiters used by the **s**, **global**, and **v** commands be accepted as well.

Preserve

The **preserve** command does not historically cause the file to be considered unmodified for the purposes of future commands that may exit the editor. IEEE Std 1003.1-200x requires conformance to historical practice.

Historical documentation stated that mail was not sent to the user when preserve was executed; however, historical implementations did send mail in this case. IEEE Std 1003.1-200x requires conformance to the historical implementations.

Print

The writing of NUL by the **print** command is not specified as a special case because the standard developers did not want to require *ex* to support NUL characters. Historically, characters were displayed using the ARPA standard mappings, which are as follows:

1. Printable characters are left alone.
2. Control characters less than \177 are represented as `'^'` followed by the character offset from the `'@'` character in the ASCII map; for example, \007 is represented as `'^G'`.
3. \177 is represented as `'^'` followed by `'?'`.

The display of characters having their eighth bit set was less standard. Existing implementations use hex (0x00), octal (\000), and a meta-bit display. (The latter displayed bytes that had their eighth bit set as the two characters `"M-"` followed by the seven-bit display as described above.) The latter probably has the best claim to historical practice because it was used for the **–v** option of 4 BSD and 4 BSD-derived versions of the *cat* utility since 1980.

No specific display format is required by IEEE Std 1003.1-200x.

Explicit dependence on the ASCII character set has been avoided where possible, hence the use of the phrase an ‘‘implementation-defined multi-character sequence’’ for the display of non-printable characters in preference to the historical usage of, for instance, `"^I"` for the <tab>. Implementations are encouraged to conform to historical practice in the absence of any strong reason to diverge.

Historically, all *ex* commands beginning with the letter `'p'` could be entered using capitalized versions of the commands; for example, **P[rint]**, **Pre[serve]**, and **Pu[t]** were all valid command names. IEEE Std 1003.1-200x permits, but does not require, this historical practice because capital forms of the commands are used by some implementations for other purposes.

Put

Historically, an *ex* **put** command, executed from open or visual mode, was the same as the open or visual mode **P** command, if the buffer was named and was cut in character mode, and the same as the **p** command if the buffer was named and cut in line mode. If the unnamed buffer was the source of the text, the entire line from which the text was taken was usually **put**, and the buffer was handled as if in line mode, but it was possible to get extremely anomalous behavior. In addition, using the **Q** command to switch into *ex* mode, and then doing a **put** often resulted in errors as well, such as appending text that was unrelated to the (supposed) contents of the buffer. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit these behaviors. All *ex* **put** commands are required to operate in line mode, and the contents of the buffers are not altered by changing the mode of the editor.

Read

Historically, an *ex* **read** command executed from open or visual mode, executed in an empty file, left an empty line as the first line of the file. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior. Historically, a **read** in open or visual mode from a program left the cursor at the last line read in, not the first. For consistency, IEEE Std 1003.1-200x does not permit this behavior.

Historical implementations of *ex* were unable to undo **read** commands that read from the output of a program. For consistency, IEEE Std 1003.1-200x does not permit this behavior.

Historically, the *ex* and *vi* message after a successful **read** or **write** command specified ‘‘characters’’, not ‘‘bytes’’. IEEE Std 1003.1-200x requires that the number of bytes be displayed, not the number of characters, because it may be difficult in multi-byte implementations to determine the number of characters read. Implementations are encouraged to clarify the message displayed to the user.

Historically, reads were not permitted on files other than type regular, except that FIFO files could be read (probably only because they did not exist when *ex* and *vi* were originally written). Because the historical *ex* evaluated **read!** and **read !** equivalently, there can be no optional way to force the read. IEEE Std 1003.1-200x permits, but does not require, this behavior.

Recover

Some historical implementations of the editor permitted users to recover the edit buffer contents from a previous edit session, and then exit without saving those contents (or explicitly discarding them). The intent of IEEE Std 1003.1-200x in requiring that the edit buffer be treated as already modified is to prevent this user error.

Rewind

Historical implementations supported the **rewind** command when the user was editing the first file in the list; that is, the file that the **rewind** command would edit. IEEE Std 1003.1-200x requires conformance to historical practice.

Substitute

Historically, *ex* accepted an **r** option to the **s** command. The effect of the **r** option was to use the last regular expression used in any command as the pattern, the same as the **˜** command. The **r** option is not required by IEEE Std 1003.1-200x. Historically, the **c** and **g** options were toggled; for example, the command **:s/abc/def/** was the same as **s/abc/def/ccccgggg**. For simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

The tilde command is often used to replace the last search RE. For example, in the sequence:

```
s/red/blue/
/green
~
```

the **˜** command is equivalent to:

```
s/green/blue/
```

Historically, *ex* accepted all of the following forms:

```
s/abc/def/
s/abc/def
s/abc/
s/abc
```

IEEE Std 1003.1-200x requires conformance to this historical practice.

The **s** command presumes that the `'^'` character only occupies a single column in the display. Much of the *ex* and *vi* specification presumes that the <space> only occupies a single column in the display. There are no known character sets for which this is not true.

Historically, the final column position for the substitute commands was based on previous column movements; a search for a pattern followed by a substitution would leave the column position unchanged, while a 0 command followed by a substitution would change the column position to the first non-<blank>. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that the final column position always be set to the first non-<blank>.

Set

Historical implementations redisplayed all of the options for each occurrence of the **all** keyword. IEEE Std 1003.1-200x permits, but does not require, this behavior.

Tag

No requirement is made as to where *ex* and *vi* shall look for the file referenced by the tag entry. Historical practice has been to look for the path found in the **tags** file, based on the current directory. A useful extension found in some implementations is to look based on the directory containing the tags file that held the entry, as well. No requirement is made as to which reference for the tag in the tags file is used. This is deliberate, in order to permit extensions such as multiple entries in a tags file for a tag.

Because users often specify many different tags files, some of which need not be relevant or exist at any particular time, IEEE Std 1003.1-200x requires that error messages about problem tags files be displayed only if the requested tag is not found, and then, only once for each time that the **tag** edit option is changed.

The requirement that the current edit buffer be unmodified is only necessary if the file indicated by the tag entry is not the same as the current file (as defined by the current pathname).

Historically, the file would be reloaded if the filename had changed, as well as if the filename was different from the current pathname. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior, requiring that the name be the only factor in the decision.

Historically, *vi* only searched for tags in the current file from the current cursor to the end of the file, and therefore, if the **wrapscan** option was not set, tags occurring before the current cursor were not found. IEEE Std 1003.1-200x considers this a bug, and implementations are required to search for the first occurrence in the file, regardless.

Undo

The **undo** description deliberately uses the word ‘‘modified’’. The **undo** command is not intended to undo commands that replace the contents of the edit buffer, such as **edit**, **next**, **tag**, or **recover**.

Cursor positioning after the **undo** command was inconsistent in the historical *vi*, sometimes attempting to restore the original cursor position (**global**, **undo**, and **v** commands), and sometimes, in the presence of maps, placing the cursor on the last line added or changed instead of the first. IEEE Std 1003.1-200x requires a simplified behavior for consistency and simplicity of specification.

Version

The **version** command cannot be exactly specified since there is no widely-accepted definition of what the version information should contain. Implementations are encouraged to do something reasonably intelligent.

Write

Historically, the *ex* and *vi* message after a successful **read** or **write** command specified ‘‘characters’’, not ‘‘bytes’’. IEEE Std 1003.1-200x requires that the number of bytes be displayed, not the number of characters because it may be difficult in multi-byte implementations to determine the number of characters written. Implementations are encouraged to clarify the message displayed to the user.

Implementation-defined tests are permitted so that implementations can make additional checks; for example, for locks or file modification times.

Historically, attempting to append to a nonexistent file caused an error. It has been left unspecified in IEEE Std 1003.1-200x to permit implementations to let the **write** succeed, so that the append semantics are similar to those of the historical *csh*.

Historical *vi* permitted empty edit buffers to be written. However, since the way *vi* got around dealing with ‘‘empty’’ files was to always have a line in the edit buffer, no matter what, it wrote them as files of a single, empty line. IEEE Std 1003.1-200x does not permit this behavior.

Historically, *ex* restored standard output and standard error to their values as of when *ex* was invoked, before writes to programs were performed. This could disturb the terminal configuration as well as be a security issue for some terminals. IEEE Std 1003.1-200x does not permit this, requiring that the program output be captured and displayed as if by the *ex* **print** command.

Adjust Window

Historically, the line count was set to the value of the **scroll** option if the type character was end-of-file. This feature was broken on most historical implementations long ago, however, and is not documented anywhere. For this reason, IEEE Std 1003.1-200x is resolutely silent.

Historically, the **z** command was <blank>-sensitive and **z +** and **z –** did different things than **z+** and **z–** because the type could not be distinguished from a flag. (The commands **z .** and **z =** were historically invalid.) IEEE Std 1003.1-200x requires conformance to this historical practice.

Historically, the **z** command was further <blank>-sensitive in that the *count* could not be <blank>-delimited; for example, the commands **z= 5** and **z– 5** were also invalid. Because the *count* is not ambiguous with respect to either the type character or the flags, this is not permitted by IEEE Std 1003.1-200x.

Escape

Historically, *ex* filter commands only read the standard output of the commands, letting standard error appear on the terminal as usual. The *vi* utility, however, read both standard output and standard error. IEEE Std 1003.1-200x requires the latter behavior for both *ex* and *vi*, for consistency.

Shift Left and Shift Right

Historically, it was possible to add shift characters to increase the effect of the command; for example, **<<<** outdented (or **>>>** indented) the lines 3 levels of indentation instead of the default 1. IEEE Std 1003.1-200x requires conformance to historical practice.

<control>-D

Historically, the <control>-D command erased the prompt, providing the user with an unbroken presentation of lines from the edit buffer. This is not required by IEEE Std 1003.1-200x; implementations are encouraged to provide it if possible. Historically, the <control>-D command took, and then ignored, a *count*. IEEE Std 1003.1-200x does not permit this behavior.

Write Line Number

Historically, the *ex* **=** command, when executed in *ex* mode in an empty edit buffer, reported 0, and from open or visual mode, reported 1. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Execute

Historically, *ex* did not correctly handle the inclusion of text input commands (that is, **append**, **insert**, and **change**) in executed buffers. IEEE Std 1003.1-200x does not permit this exclusion for consistency.

Historically, the logical contents of the buffer being executed did not change if the buffer itself were modified by the commands being executed; that is, buffer execution did not support self-modifying code. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the **@** command took a range of lines, and the **@** buffer was executed once per line, with the current line (`'.'`) set to each specified line. IEEE Std 1003.1-200x requires conformance to historical practice.

Some historical implementations did not notice if errors occurred during buffer execution. This, coupled with the ability to specify a range of lines for the *ex* **@** command, makes it trivial to cause them to drop core. IEEE Std 1003.1-200x requires that implementations stop buffer execution if any error occurs, if the specified line doesn't exist, or if the contents of the edit buffer itself are replaced (for example, the buffer executes the *ex* **:edit** command).

Regular Expressions in ex

Historical practice is that the characters in the replacement part of the last **s** command—that is, those matched by entering a `'~'` in the regular expression—were not further expanded by the regular expression engine. So, if the characters contained the string `"a.,"` they would match `'a'` followed by `".,"` and not `'a'` followed by any character. IEEE Std 1003.1-200x requires conformance to historical practice.

Edit Options in ex

The following paragraphs describe the historical behavior of some edit options that were not, for whatever reason, included in IEEE Std 1003.1-200x. Implementations are strongly encouraged to only use these names if the functionality described here is fully supported.

extended	The **extended** edit option has been used in some implementations of *vi* to provide extended regular expressions instead of basic regular expressions This option was omitted from IEEE Std 1003.1-200x because it is not widespread historical practice.
flash	The **flash** edit option historically caused the screen to flash instead of beeping on error. This option was omitted from IEEE Std 1003.1-200x because it is not found in some historical implementations.
hardtabs	The **hardtabs** edit option historically defined the number of columns between hardware tab settings. This option was omitted from IEEE Std 1003.1-200x because it was believed to no longer be generally useful.
modeline	The **modeline** (sometimes named **modelines**) edit option historically caused *ex* or *vi* to read the five first and last lines of the file for editor commands. This option is a security problem, and vendors are strongly encouraged to delete it from historical implementations.
open	The **open** edit option historically disallowed the *ex* **open** and **visual** commands. This edit option was omitted because these commands are required by IEEE Std 1003.1-200x.
optimize	The **optimize** edit option historically expedited text throughput by setting the terminal to not do automatic carriage returns when printing more than one logical line of output. This option was omitted from IEEE Std 1003.1-200x because it was intended for terminals without addressable cursors, which are rarely, if ever, still used.
ruler	The **ruler** edit option has been used in some implementations of *vi* to present a current row/column ruler for the user. This option was omitted from IEEE Std 1003.1-200x because it is not widespread historical practice.
sourceany	The **sourceany** edit option historically caused *ex* or *vi* to source start-up files that were owned by users other than the user running the editor. This option is a security problem, and vendors are strongly encouraged to remove it from their implementations.
timeout	The **timeout** edit option historically enabled the (now standard) feature of only waiting for a short period before returning keys that could be part of a macro. This feature was omitted from IEEE Std 1003.1-200x because its behavior is now standard, it is not widely useful, and it was rarely documented.

verbose The **verbose** edit option has been used in some implementations of *vi* to cause *vi* to output error messages for common errors; for example, attempting to move the cursor past the beginning or end of the line instead of only alerting the screen. (The historical *vi* only alerted the terminal and presented no message for such errors. The historical editor option **terse** did not select when to present error messages, it only made existing error messages more or less verbose.) This option was omitted from IEEE Std 1003.1-200x because it is not widespread historical practice; however, implementors are encouraged to use it if they wish to provide error messages for naive users.

wraplen The **wraplen** edit option has been used in some implementations of *vi* to specify an automatic margin measured from the left margin instead of from the right margin. This is useful when multiple screen sizes are being used to edit a single file. This option was omitted from IEEE Std 1003.1-200x because it is not widespread historical practice; however, implementors are encouraged to use it if they add this functionality.

autoindent, ai

Historically, the command **0a** did not do any autoindentation, regardless of the current indentation of line 1. IEEE Std 1003.1-200x requires that any indentation present in line 1 be used.

autoprint, ap

Historically, the **autoprint** edit option was not completely consistent or based solely on modifications to the edit buffer. Exceptions were the **read** command (when reading from a file, but not from a filter), the **append**, **change**, **insert**, **global**, and **v** commands, all of which were not affected by **autoprint**, and the **tag** command, which was affected by **autoprint**. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the **autoprint** option only applied to the last of multiple commands entered using vertical-bar delimiters; for example, **delete** <newline> was affected by **autoprint**, but **delete | version** <newline> was not. IEEE Std 1003.1-200x requires conformance to historical practice.

autowrite, aw

Appending the '!' character to the *ex* **next** command to avoid performing an automatic write was not supported in historical implementations. IEEE Std 1003.1-200x requires that the behavior match the other *ex* commands for consistency.

ignorecase, ic

Historical implementations of case-insensitive matching (the **ignorecase** edit option) lead to counterintuitive situations when uppercase characters were used in range expressions. Historically, the process was as follows:

1. Take a line of text from the edit buffer.
2. Convert uppercase to lowercase in text line.
3. Convert uppercase to lowercase in regular expressions, except in character class specifications.
4. Match regular expressions against text.

This would mean that, with **ignorecase** in effect, the text:

```
The cat sat on the mat
```

would be matched by

```
/ˆthe/
```

but not by:

```
/ˆ[A–Z]he/
```

For consistency with other commands implementing regular expressions, IEEE Std 1003.1-200x does not permit this behavior.

paragraphs, para

Earlier versions of IEEE Std 1003.1-200x made the default **paragraphs** and **sections** edit options implementation-defined, arguing they were historically oriented to the UNIX system *troff* text formatter, and a ‘‘portable user’’ could use the **{**, **}**, **[[**, **]]**, **(**, and **)** commands in open or visual mode and have the cursor stop in unexpected places. IEEE Std 1003.1-200x specifies their values in the POSIX locale because the unusual grouping (they only work when grouped into two characters at a time) means that they cannot be used for general purpose movement, regardless.

readonly

Implementations are encouraged to provide the best possible information to the user as to the read-only status of the file, with the exception that they should not consider the current special privileges of the process. This provides users a safety net because they must force the overwrite of read-only files, even when running with additional privileges.

The **readonly** edit option specification largely conforms to historical practice. The only difference is that historical implementations did not notice that the user had set the **readonly** edit option in cases where the file was already marked read-only for some reason, and would therefore reinitialize the **readonly** edit option the next time the contents of the edit buffer were replaced. This behavior is disallowed by IEEE Std 1003.1-200x.

report

The requirement that lines copied to a buffer interact differently than deleted lines is historical practice. For example, if the **report** edit option is set to 3, deleting 3 lines will cause a report to be written, but 4 lines must be copied before a report is written.

The requirement that the *ex* **global**, **v**, **open**, **undo**, and **visual** commands present reports based on the total number of lines added or deleted during the command execution, and that commands executed by the **global** and **v** commands not present reports, is historical practice. IEEE Std 1003.1-200x extends historical practice by requiring that buffer execution be treated similarly. The reasons for this are two-fold. Historically, only the report by the last command executed from the buffer would be seen by the user, as each new report would overwrite the last. In addition, the standard developers believed that buffer execution had more in common with **global** and **v** commands than it did with other *ex* commands, and should behave similarly, for consistency and simplicity of specification.

showmatch, sm

The length of time the cursor spends on the matching character is unspecified because the timing capabilities of systems are often inexact and variable. The time should be long enough for the user to notice, but not long enough for the user to become annoyed. Some implementations of *vi* have added a **matchtime** option that permits users to set the number of 0,1 second intervals the cursor pauses on the matching character.

showmode

The **showmode** option has been used in some historical implementations of *ex* and *vi* to display the current editing mode when in open or visual mode. The editing modes have generally included ''command'' and ''input'', and sometimes other modes such as ''replace'' and ''change''. The string was usually displayed on the bottom line of the screen at the far right-hand corner. In addition, a preceding `'*'` character often denoted if the contents of the edit buffer had been modified. The latter display has sometimes been part of the **showmode** option, and sometimes based on another option. This option was not available in the 4 BSD historical implementation of *vi*, but was viewed as generally useful, particularly to novice users, and is required by IEEE Std 1003.1-200x.

The **smd** shorthand for the **showmode** option was not present in all historical implementations of the editor. IEEE Std 1003.1-200x requires it, for consistency.

Not all historical implementations of the editor displayed a mode string for command mode, differentiating command mode from text input mode by the absence of a mode string. IEEE Std 1003.1-200x permits this behavior for consistency with historical practice, but implementations are encouraged to provide a display string for both modes.

slowopen

Historically the **slowopen** option was automatically set if the terminal baud rate was less than 1 200 baud, or if the baud rate was 1 200 baud and the **redraw** option was not set. The **slowopen** option had two effects. First, when inserting characters in the middle of a line, characters after the cursor would not be pushed ahead, but would appear to be overwritten. Second, when creating a new line of text, lines after the current line would not be scrolled down, but would appear to be overwritten. In both cases, ending text input mode would cause the screen to be refreshed to match the actual contents of the edit buffer. Finally, terminals that were sufficiently intelligent caused the editor to ignore the **slowopen** option. IEEE Std 1003.1-200x permits most historical behavior, extending historical practice to require **slowopen** behaviors if the edit option is set by the user.

tags

The default path for tags files is left unspecified as implementations may have their own **tags** implementations that do not correspond to the historical ones. The default **tags** option value should probably at least include the file **./tags**.

term

Historical implementations of *ex* and *vi* ignored changes to the **term** edit option after the initial terminal information was loaded. This is permitted by IEEE Std 1003.1-200x; however, implementations are encouraged to permit the user to modify their terminal type at any time.

terse

Historically, the **terse** edit option optionally provided a shorter, less descriptive error message, for some error messages. This is permitted, but not required, by IEEE Std 1003.1-200x. Historically, most common visual mode errors (for example, trying to move the cursor past the end of a line) did not result in an error message, but simply alerted the terminal. Implementations wishing to provide messages for novice users are urged to do so based on the **edit** option **verbose**, and not **terse**.

window

In historical implementations, the default for the **window** edit option was based on the baud rate as follows:

1. If the baud rate was less than 1 200, the **edit** option **w300** set the window value; for example, the line:

   ```
   set w300=12
   ```

   would set the window option to 12 if the baud rate was less than 1 200.
2. If the baud rate was equal to 1 200, the **edit** option **w1200** set the window value.
3. If the baud rate was greater than 1 200, the **edit** option **w9600** set the window value.

The **w300**, **w1200**, and **w9600** options do not appear in IEEE Std 1003.1-200x because of their dependence on specific baud rates.

In historical implementations, the size of the window displayed by various commands was related to, but not necessarily the same as, the **window** edit option. For example, the size of the window was set by the *ex* command **visual 10**, but it did not change the value of the **window** edit option. However, changing the value of the **window** edit option did change the number of lines that were displayed when the screen was repainted. IEEE Std 1003.1-200x does not permit this behavior in the interests of consistency and simplicity of specification, and requires that all commands that change the number of lines that are displayed do it by setting the value of the **window** edit option.

wrapmargin, wm

Historically, the **wrapmargin** option did not affect maps inserting characters that also had associated *count*s; for example **:map K 5aABC DEF**. Unfortunately, there are widely used maps that depend on this behavior. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Historically, **wrapmargin** was calculated using the column display width of all characters on the screen. For example, an implementation using `"^I"` to represent <tab>s when the **list** edit option was set, where `'^'` and `'I'` each took up a single column on the screen, would calculate the **wrapmargin** based on a value of 2 for each <tab>. The **number** edit option similarly changed the effective length of the line as well. IEEE Std 1003.1-200x requires conformance to historical practice.

FUTURE DIRECTIONS

None.

SEE ALSO

ed, *sed*, *stty*, *vi*, the System Interfaces volume of IEEE Std 1003.1-200x, *access*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

The FUTURE DIRECTIONS section is added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The obsolescent SYNOPSIS is removed, removing the **+***command* and – options.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the *map* command description, the sequence **#***digit* is added.
- The **directory**, **edcompatible**, **redraw**, and **slowopen** edit options are added.

The *ex* utility is extensively changed for alignment with the IEEE P1003.2b draft standard. This includes changes as a result of the IEEE PASC Interpretations 1003.2 #31, #38, #49, #50, #51, #52, #55, #56, #57, #61, #62, #63, #64, #65, and #78.

NAME

expand — convert tabs to spaces

SYNOPSIS

UP `expand [−t tablist][file ...]`

DESCRIPTION

The *expand* utility shall write files or the standard input to the standard output with <tab>s replaced with one or more <space>s needed to pad to the next tab stop. Any <backspace>s shall be copied to the output and cause the column position count for tab stop calculations to be decremented; the column position count shall not be decremented below zero.

OPTIONS

The *expand* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−t *tablist* Specify the tab stops. The application shall ensure that the argument *tablist* consists of either a single positive decimal integer or a list of tabstops. If a single number is given, tabs shall be set that number of column positions apart instead of the default 8.

If a list of tabstops is given, the application shall ensure that it consists of a list of two or more positive decimal integers, separated by <blank>s or commas, in ascending order. The tabs shall be set at those specific column positions. Each tab stop *N* shall be an integer value greater than zero, and the list is in strictly ascending order. This is taken to mean that, from the start of a line of output, tabbing to position *N* shall cause the next character output to be in the (*N*+1)th column position on that line.

In the event of *expand* having to process a <tab> at a position beyond the last of those specified in a multiple tab-stop list, the <tab> shall be replaced by a single <space> in the output.

OPERANDS

The following operand shall be supported:

file The pathname of a text file to be used as input.

STDIN

See the INPUT FILES section.

INPUT FILES

Input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *expand*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the processing of <tab>s and <space>s, and for the determination of the width in column positions each character would occupy on an output device.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be equivalent to the input files with <tab>s converted into the appropriate number of <space>s.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion
>0	An error occurred.

CONSEQUENCES OF ERRORS

The *expand* utility shall terminate with an error message and non-zero exit status upon encountering difficulties accessing one of the *file* operands.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *expand* utility is useful for preprocessing text files (before sorting, looking at specific columns, and so on) that contain <tab>s.

See the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.103, Column Position.

The *tablist* option-argument consists of integers in ascending order. Utility Syntax Guideline 8 mandates that *expand* shall accept the integers (within the single argument) separated using either commas or <blank>s.

FUTURE DIRECTIONS

None.

SEE ALSO

tabs, *unexpand*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

The obsolescent SYNOPSIS is removed.

The *LC_CTYPE* environment variable description is updated to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

expr — evaluate arguments as an expression

SYNOPSIS

`expr` *`operand`*

DESCRIPTION

The *expr* utility shall evaluate an expression and write the result to standard output.

OPTIONS

None.

OPERANDS

The single expression evaluated by *expr* shall be formed from the operands, as described in the EXTENDED DESCRIPTION section. The application shall ensure that each of the expression operator symbols:

```
(   )   |   &   =   >   >=   <   <=   !=   +   –   *   /   %   :
```

and the symbols *integer* and *string* in the table are provided as separate arguments to *expr*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *expr*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions and by the string comparison operators.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments) and the behavior of character classes within regular expressions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *expr* utility shall evaluate the expression and write the result, followed by a <newline>, to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The formation of the expression to be evaluated is shown in the following table. The symbols *expr*, *expr1*, and *expr2* represent expressions formed from *integer* and *string* symbols and the expression operator symbols (all separate arguments) by recursive application of the constructs described in the table. The expressions are listed in order of increasing precedence, with equal-precedence operators grouped between horizontal lines. All of the operators shall be left-associative.

Expression	Description
expr1 | _expr2_	Returns the evaluation of _expr1_ if it is neither null nor zero; otherwise, returns the evaluation of _expr2_ if it is not null; otherwise, zero.
expr1 & _expr2_	Returns the evaluation of _expr1_ if neither expression evaluates to null or zero; otherwise, returns zero.
	Returns the result of a decimal integer comparison if both arguments are integers; otherwise, returns the result of a string comparison using the locale-specific collation sequence. The result of each comparison is 1 if the specified relationship is true, or 0 if the relationship is false.
expr1 = _expr2_	Equal.
expr1 > _expr2_	Greater than.
expr1 >= _expr2_	Greater than or equal.
expr1 < _expr2_	Less than.
expr1 <= _expr2_	Less than or equal.
expr1 != _expr2_	Not equal.
expr1 + _expr2_	Addition of decimal integer-valued arguments.
expr1 − _expr2_	Subtraction of decimal integer-valued arguments.
expr1 * _expr2_	Multiplication of decimal integer-valued arguments.
expr1 / _expr2_	Integer division of decimal integer-valued arguments, producing an integer result.
expr1 % _expr2_	Remainder of integer division of decimal integer-valued arguments.
expr1 : _expr2_	Matching expression; see below.
(_expr_)	Grouping symbols. Any expression can be placed within parentheses. Parentheses can be nested to a depth of {EXPR_NEST_MAX}.
integer	An argument consisting only of an (optional) unary minus followed by digits.
string	A string argument; see below.

Matching Expression

The `':'` matching operator shall compare the string resulting from the evaluation of *expr1* with the regular expression pattern resulting from the evaluation of *expr2*. Regular expression syntax shall be that defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions, except that all patterns are anchored to the beginning of the string (that is, only sequences starting at the first character of a string are matched by the regular expression) and, therefore, it is unspecified whether `'^'` is a special character in that context. Usually, the matching operator shall return a string representing the number of characters matched (`'0'` on failure). Alternatively, if the pattern contains at least one regular expression subexpression `"[\(...\)]"`, the string corresponding to `"\1"` shall be returned.

String Operand

A string argument is an argument that cannot be identified as an *integer* argument or as one of the expression operator symbols shown in the OPERANDS section.

The use of string arguments **length**, **substr**, **index**, or **match** produces unspecified results.

EXIT STATUS

The following exit values shall be returned:

0 The *expression* evaluates to neither null nor zero.

1 The *expression* evaluates to null or zero.

2 Invalid *expression*.

>2 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

After argument processing by the shell, *expr* is not required to be able to tell the difference between an operator and an operand except by the value. If `"$a"` is `'='`, the command:

```
expr $a = '='
```

looks like:

```
expr = = =
```

as the arguments are passed to *expr* (and they all may be taken as the `'='` operator). The following works reliably:

```
expr X$a = X=
```

Also note that this volume of IEEE Std 1003.1-200x permits implementations to extend utilities. The *expr* utility permits the integer arguments to be preceded with a unary minus. This means that an integer argument could look like an option. Therefore, the conforming application must employ the `"--"` construct of Guideline 10 of the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines to protect its operands if there is any chance the first operand might be a negative integer (or any string with a leading minus).

EXAMPLES

The *expr* utility has a rather difficult syntax:

- Many of the operators are also shell control operators or reserved words, so they have to be escaped on the command line.

- Each part of the expression is composed of separate arguments, so liberal usage of <blank>s is required. For example:

Invalid	**Valid**
`expr 1+2`	`expr 1 + 2`
`expr "1 + 2"`	`expr 1 + 2`
`expr 1 + (2 * 3)`	`expr 1 + \( 2 \* 3 \)`

In many cases, the arithmetic and string features provided as part of the shell command language are easier to use than their equivalents in *expr*. Newly written scripts should avoid *expr* in favor of the new features within the shell; see Section 2.5 (on page 2235) and Section 2.6.4 (on page 2243).

The following command:

```
a=$(expr $a + 1)
```

adds 1 to the variable *a*.

The following command, for `"$a"` equal to either **/usr/abc/file** or just **file**:

```
expr $a : '.*/\(.*\)' \| $a
```

returns the last segment of a pathname (that is, **file**). Applications should avoid the character `'/'` used alone as an argument: *expr* may interpret it as the division operator.

The following command:

```
expr "//$a" : '.*/\(.*\)'
```

is a better representation of the previous example. The addition of the `"//"` characters eliminates any ambiguity about the division operator and simplifies the whole expression. Also note that pathnames may contain characters contained in the *IFS* variable and should be quoted to avoid having `"$a"` expand into multiple arguments.

The following command:

```
expr "$VAR" : '.*'
```

returns the number of characters in *VAR*.

RATIONALE

In an early proposal, EREs were used in the matching expression syntax. This was changed to BREs to avoid breaking historical applications.

The use of a leading circumflex in the BRE is unspecified because many historical implementations have treated it as a special character, despite their system documentation. For example:

```
expr foo : ^foo     expr ^foo : ^foo
```

return 3 and 0, respectively, on those systems; their documentation would imply the reverse. Thus, the anchoring condition is left unspecified to avoid breaking historical scripts relying on this undocumented feature.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.6.4

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The *expr* utility is aligned with the IEEE P1003.2b draft standard, to include resolution of IEEE PASC Interpretation 1003.2 #104.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

false — return false value

SYNOPSIS

`false`

DESCRIPTION

The *false* utility shall return with a non-zero exit code.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The *false* utility always shall exit with a value other than zero.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

true

CHANGE HISTORY

First released in Issue 2.

NAME

fc — process the command history list

SYNOPSIS

UP

```
fc [−r][−e editor] [first[last]]
fc −l[−nr] [first[last]]
fc −s[old=new][first]
```

DESCRIPTION

The *fc* utility shall list, or shall edit and re-execute, commands previously entered to an interactive *sh*.

The command history list shall reference commands by number. The first number in the list is selected arbitrarily. The relationship of a number to its command shall not change except when the user logs in and no other process is accessing the list, at which time the system may reset the numbering to start the oldest retained command at another number (usually 1). When the number reaches an implementation-defined upper limit, which shall be no smaller than the value in *HISTSIZE* or 32 767 (whichever is greater), the shell may wrap the numbers, starting the next command with a lower number (usually 1). However, despite this optional wrapping of numbers, *fc* shall maintain the time-ordering sequence of the commands. For example, if four commands in sequence are given the numbers 32 766, 32 767, 1 (wrapped), and 2 as they are executed, command 32 767 is considered the command previous to 1, even though its number is higher.

When commands are edited (when the **−l** option is not specified), the resulting lines shall be entered at the end of the history list and then re-executed by *sh*. The *fc* command that caused the editing shall not be entered into the history list. If the editor returns a non-zero exit status, this shall suppress the entry into the history list and the command re-execution. Any command line variable assignments or redirection operators used with *fc* shall affect both the *fc* command itself as well as the command that results; for example:

```
fc −s −− −1 2>/dev/null
```

reinvokes the previous command, suppressing standard error for both *fc* and the previous command.

OPTIONS

The *fc* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−e *editor*	Use the editor named by *editor* to edit the commands. The *editor* string is a utility name, subject to search via the *PATH* variable (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables). The value in the *FCEDIT* variable shall be used as a default when **−e** is not specified. If *FCEDIT* is null or unset, *ed* shall be used as the editor.
−l	(The letter ell.) List the commands rather than invoking an editor on them. The commands shall be written in the sequence indicated by the *first* and *last* operands, as affected by **−r**, with each command preceded by the command number.
−n	Suppress command numbers when listing with **−l**.
−r	Reverse the order of the commands listed (with **−l**) or edited (with neither **−l** nor **−s**).

−s Reexecute the command without invoking an editor.

OPERANDS

The following operands shall be supported:

first, *last*

Select the commands to list or edit. The number of previous commands that can be accessed shall be determined by the value of the *HISTSIZE* variable. The value of *first* or *last* or both shall be one of the following:

[+]*number* A positive number representing a command number; command numbers can be displayed with the **−l** option.

−*number* A negative decimal number representing the command that was executed *number* of commands previously. For example, −1 is the immediately previous command.

string A string indicating the most recently entered command that begins with that string. If the *old*=*new* operand is not also specified with **−s**, the string form of the *first* operand cannot contain an embedded equal sign.

When the synopsis form with **−s** is used:

- If *first* is omitted, the previous command shall be used.

For the synopsis forms without **−s**:

- If *last* is omitted, *last* shall default to the previous command when **−l** is specified; otherwise, it shall default to *first*.
- If *first* and *last* are both omitted, the previous 16 commands shall be listed or the previous single command shall be edited (based on the **−l** option).
- If *first* and *last* are both present, all of the commands from *first* to *last* shall be edited (without **−l**) or listed (with **−l**). Editing multiple commands shall be accomplished by presenting to the editor all of the commands at one time, each command starting on a new line. If *first* represents a newer command than *last*, the commands shall be listed or edited in reverse sequence, equivalent to using **−r**. For example, the following commands on the first line are equivalent to the corresponding commands on the second:

```
fc −r 10 20    fc    30 40
fc    20 10    fc −r 40 30
```

- When a range of commands is used, it shall not be an error to specify *first* or *last* values that are not in the history list; *fc* shall substitute the value representing the oldest or newest command in the list, as appropriate. For example, if there are only ten commands in the history list, numbered 1 to 10:

```
fc −l
fc 1 99
```

  shall list and edit, respectively, all ten commands.

old=*new* Replace the first occurrence of string *old* in the commands to be re-executed by the string *new*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *fc*:

FCEDIT	This variable, when expanded by the shell, shall determine the default value for the **−e** *editor* option's *editor* option-argument. If *FCEDIT* is null or unset, *ed* shall be used as the editor.
HISTFILE	Determine a pathname naming a command history file. If the *HISTFILE* variable is not set, the shell may attempt to access or create a file **.sh_history** in the directory referred to by the *HOME* environment variable. If the shell cannot obtain both read and write access to, or create, the history file, it shall use an unspecified mechanism that allows the history to operate properly. (References to history ''file'' in this section shall be understood to mean this unspecified mechanism in such cases.) An implementation may choose to access this variable only when initializing the history file; this initialization shall occur when *fc* or *sh* first attempt to retrieve entries from, or add entries to, the file, as the result of commands issued by the user, the file named by the *ENV* variable, or implementation-defined system start-up files. In some historical shells, the history file is initialized just after the *ENV* file has been processed. Therefore, it is implementation-defined whether changes made to *HISTFILE* after the history file has been initialized are effective. Implementations may choose to disable the history list mechanism for users with appropriate privileges who do not set *HISTFILE*; the specific circumstances under which this occurs are implementation-defined. If more than one instance of the shell is using the same history file, it is unspecified how updates to the history file from those shells interact. As entries are deleted from the history file, they shall be deleted oldest first. It is unspecified when history file entries are physically removed from the history file.
HISTSIZE	Determine a decimal number representing the limit to the number of previous commands that are accessible. If this variable is unset, an unspecified default greater than or equal to 128 shall be used. The maximum number of commands in the history list is unspecified, but shall be at least 128. An implementation may choose to access this variable only when initializing the history file, as described under *HISTFILE*. Therefore, it is unspecified whether changes made to *HISTSIZE* after the history file has been initialized are effective.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–l** option is used to list commands, the format of each command in the list shall be as follows:

```
"%d\t%s\n", <line number>, <command>
```

If both the **–l** and **–n** options are specified, the format of each command shall be:

```
"\t%s\n", <command>
```

If the <*command*> consists of more than one line, the lines after the first shall be displayed as:

```
"\t%s\n", <continued-command>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion of the listing.
>0	An error occurred.

Otherwise, the exit status shall be that of the commands executed by *fc*.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since editors sometimes use file descriptors as integral parts of their editing, redirecting their file descriptors as part of the *fc* command can produce unexpected results. For example, if *vi* is the *FCEDIT* editor, the command:

```
fc –s | more
```

does not work correctly on many systems.

Users on windowing systems may want to have separate history files for each window by setting *HISTFILE* as follows:

```
HISTFILE=$HOME/.sh_hist$$
```

EXAMPLES

None.

RATIONALE

This utility is based on the *fc* built-in of the KornShell.

An early proposal specified the **–e** option as **[–e** *editor* **[***old*= *new* **]]**, which is not historical practice. Historical practice in *fc* of either **[–e** *editor***]** or **[–e** – **[** *old*= *new* **]]** is acceptable, but not both together. To clarify this, a new option **–s** was introduced replacing the **[–e –]**. This resolves the conflict and makes *fc* conform to the Utility Syntax Guidelines.

HISTFILE	Some implementations of the KornShell check for the superuser and do not create a history file unless *HISTFILE* is set. This is done primarily to avoid creating unlinked files in the root file system when logging in during single-user mode. *HISTFILE* must be set for the superuser to have history.
HISTSIZE	Needed to limit the size of history files. It is the intent of the standard developers that when two shells share the same history file, commands that are entered in one shell shall be accessible by the other shell. Because of the difficulties of synchronization over a network, the exact nature of the interaction is unspecified.

The initialization process for the history file can be dependent on the system start-up files, in that they may contain commands that effectively preempt the settings the user has for *HISTFILE* and *HISTSIZE*. For example, function definition commands are recorded in the history file. If the system administrator includes function definitions in some system start-up file called before the *ENV* file, the history file is initialized before the user can influence its characteristics. In some historical shells, the history file is initialized just after the *ENV* file has been processed. Because of these situations, the text requires the initialization process to be implementation-defined.

Consideration was given to omitting the *fc* utility in favor of the command line editing feature in *sh*. For example, in *vi* editing mode, typing `"<ESC> v"` is equivalent to:

```
EDITOR=vi fc
```

However, the *fc* utility allows the user the flexibility to edit multiple commands simultaneously (such as *fc* 10 20) and to use editors other than those supported by *sh* for command line editing.

In the KornShell, the alias **r** (‘‘re-do’’) is preset to *fc* −**e** − (equivalent to the POSIX *fc* −**s**). This is probably an easier command name to remember than *fc* (‘‘fix command’’), but it does not meet the Utility Syntax Guidelines. Renaming *fc* to *hist* or *redo* was considered, but since this description closely matches historical KornShell practice already, such a renaming was seen as gratuitous. Users are free to create aliases whenever odd historical names such as *fc*, *awk*, *cat*, *grep*, or *yacc* are standardized by POSIX.

Command numbers have no ordering effects; they are like serial numbers. The −**r** option and −*number* operand address the sequence of command execution, regardless of serial numbers. So, for example, if the command number wrapped back to 1 at some arbitrary point, there would be no ambiguity associated with traversing the wrap point. For example, if the command history were:

```
32766: echo 1
32767: echo 2
1: echo 3
```

the number −2 refers to command 32 767 because it is the second previous command, regardless of serial number.

FUTURE DIRECTIONS

None.

SEE ALSO

sh

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

In the ENVIRONMENT VARIABLES section, the text ‘‘user’s home directory’’ is updated to ‘‘directory referred to by the *HOME* environment variable’’.

NAME

fg — run jobs in the foreground

SYNOPSIS

UP
```
fg [job_id]
```

DESCRIPTION

If job control is enabled (see the description of *set* **–m**), the *fg* utility shall move a background job from the current environment (see Section 2.12 (on page 2263)) into the foreground.

Using *fg* to place a job into the foreground shall remove its process ID from the list of those ‘‘known in the current shell execution environment’’; see Section 2.9.3.1 (on page 2252).

OPTIONS

None.

OPERANDS

The following operand shall be supported:

job_id	Specify the job to be run as a foreground job. If no *job_id* operand is given, the *job_id* for the job that was most recently suspended, placed in the background or run as a background job, shall be used. The format of *job_id* is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.203, Job Control Job ID.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *fg*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *fg* utility shall write the command line of the job to standard output in the following format:

```
"%s\n", <command>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

If job control is disabled, the *fg* utility shall exit with an error and no job shall be placed in the foreground.

APPLICATION USAGE

The *fg* utility does not work as expected when it is operating in its own utility execution environment because that environment has no applicable jobs to manipulate. See the APPLICATION USAGE section for *bg* (on page 2410). For this reason, *fg* is generally implemented as a shell regular built-in.

EXAMPLES

None.

RATIONALE

The extensions to the shell specified in this volume of IEEE Std 1003.1-200x have mostly been based on features provided by the KornShell. The job control features provided by *bg*, *fg*, and *jobs* are also based on the KornShell. The standard developers examined the characteristics of the C shell versions of these utilities and found that differences exist. Despite widespread use of the C shell, the KornShell versions were selected for this volume of IEEE Std 1003.1-200x to maintain a degree of uniformity with the rest of the KornShell features selected (such as the very popular command line editing features).

FUTURE DIRECTIONS

None.

SEE ALSO

bg, *kill*, *jobs*, *wait*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

The JC marking is removed from the SYNOPSIS since job control is mandatory is this issue.

NAME

file — determine file type

SYNOPSIS

UP
```
file [−dhi][−M file][−m file] file ...
```

DESCRIPTION

The *file* utility shall perform a series of tests on each specified *file* in an attempt to classify it:

1. If the file is not a regular file, its file type shall be identified. The file types directory, FIFO, socket, block special, and character special shall be identified as such. Other implementation-defined file types may also be identified.
2. If the file is a regular file, and:
   a. The file is zero-length, it shall be identified as an empty file.
   b. The file is not zero-length, *file* shall examine an initial segment of the file and shall make a guess at identifying its contents or whether it is an executable binary file. (The answer is not guaranteed to be correct.)

If *file* does not exist, cannot be read, or its file status could not be determined, the output shall indicate that the file was processed, but that its type could not be determined.

If *file* is a symbolic link, by default the link shall be resolved and *file* shall test the type of file referenced by the symbolic link.

OPTIONS

The *file* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−d	Apply any default system tests to the file.
−h	When a symbolic link is encountered, identify the file as a symbolic link. If **−h** is not specified and *file* is a symbolic link that refers to a nonexistent file, *file* shall identify the file as a symbolic link, as if **−h** had been specified.
−i	If a file is a regular file, do not attempt to classify the type of the file further, but identify the file as specified in the STDOUT section, using a <*type*> string that contains the string `"regular file"`.
−M *file*	Specify the name of a file containing tests that shall be applied to a file in order to classify it (see the EXTENDED DESCRIPTION). No default system tests shall be applied.
−m *file*	Specify the name of a file containing tests that shall be applied to a file in order to classify it (see the EXTENDED DESCRIPTION).

If multiple instances of the **−m**, **−d**, or **−M** options are specified, the concatenation of the tests specified, in the order specified, shall be the set of tests that are applied. If a **−M** option is specified, no tests other than those specified using the **−d**, **−M**, and **−m** options shall be applied to the file. If neither the **−d** nor **−M** options are specified, any default system tests shall be applied after any tests specified using the **−m** option.

OPERANDS

The following operand shall be supported:

file	A pathname of a file to be tested.

STDIN

Not used.

INPUT FILES

The *file* can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *file*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

In the POSIX locale, the following format shall be used to identify each operand, *file* specified:

```
"%s: %s\n", <file>, <type>
```

The values for <*type*> are unspecified, except that in the POSIX locale, if *file* is identified as one of the types listed in the following table, <*type*> shall contain (but is not limited to) the corresponding string. Each space shown in the strings shall be exactly one <space>.

Table 4-8 File Utility Output Strings

If *file* is a:	**<*type*> shall contain the string:**
Directory	directory
FIFO	fifo
Socket	socket
Block special	block special
Character special	character special
Executable binary	executable
Empty regular file	empty
Symbolic link	symbolic link to
ar archive library (see *ar*)	archive
Extended *cpio* format (see *pax*)	cpio archive
Extended *tar* format (see **ustar** in *pax*)	tar archive
Shell script	commands text
C-language source	c program text
FORTRAN source	fortran program text

If *file* is identified as a symbolic link (see **−h**), the following alternative output format shall be used:

```
"%s: %s %s\n", <file>, <type>, <contents of link>"
```

If the file named by the *file* operand does not exist or cannot be read, the string `"cannot open"` shall be included as part of the <*type*> field, but this shall not be considered an error that affects the exit status. If the type of the file named by the *file* operand cannot be determined, the string `"data"` shall be included as part of the <*type*> field, but this shall not be considered an error that affects the exit status.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

A file specified as an option-argument to the **−m** or **−M** options shall contain one test per line, which shall be applied to the file. If the test succeeds, the message field of the line shall be printed and no further tests shall be applied, with the exception that tests on immediately following lines beginning with a single `'>'` character shall be applied.

Each line shall be composed of the following four <blank>-separated fields:

offset An unsigned number (optionally preceded by a single `'>'` character) specifying the *offset*, in bytes, of the value in the file that is to be compared against the *value* field of the line. If the file is shorter than the specified offset, the test shall fail.

If the *offset* begins with the character `'>'`, the test contained in the line shall not be applied to the file unless the test on the last line for which the *offset* did not begin with a `'>'` was successful. By default, the *offset* shall be interpreted as an unsigned decimal number. With a leading 0x or 0X, the *offset* shall be interpreted as a hexadecimal number; otherwise, with a leading 0, the *offset* shall be interpreted as an octal number.

type The type of the value in the file to be tested. The type shall consist of the type specification characters `c`, `d`, `f`, `s`, and `u`, specifying character, signed decimal, floating point, string, and unsigned decimal, respectively.

The *type* string shall be interpreted as the bytes from the file starting at the specified *offset* and including the same number of bytes specified by the *value* field. If insufficient bytes remain in the file past the *offset* to match the *value* field, the test shall fail.

The type specification characters `d`, `f`, and `u` can be followed by an optional unsigned decimal integer that specifies the number of bytes represented by the type. The type specification character `f` can be followed by an optional `F`, `D`, or `L`, indicating that the value is of type **float**, **double**, or **long double**, respectively. The type specification characters `d` and `u` can be followed by an optional `C`, `S`, `I`, or `L`, indicating that the value is of type **char**, **short**, **int**, or **long**, respectively.

The default number of bytes represented by the type specifiers `d`, `f`, and `u` shall correspond to their respective C-language types as follows. If the system claims conformance to the C-Language Development Utilities option, those specifiers shall correspond to the default sizes used in the *c99* utility. Otherwise, the default sizes shall be implementation-defined.

For the type specifier characters `d` and `u`, the default number of bytes shall correspond to the size of a basic integer type of the implementation. For these specifier characters, the implementation shall support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **char**, **short**, **int**, or **long**. These numbers can also be specified by an application as the characters `C`, `S`, `I`, and `L`, respectively. The byte order used when interpreting numeric values is implementation-defined, but shall correspond to the order in which a constant of the corresponding type is stored in memory on the system.

For the type specifier `f`, the default number of bytes shall correspond to the number of bytes in the basic double precision floating-point data type of the underlying implementation. The implementation shall support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **float**, **double**, and **long double**. These numbers can also be specified by an application as the characters `F`, `D`, and `L`, respectively.

All type specifiers, except for `s`, can be followed by a mask specifier of the form &*number*. The mask value shall be AND'ed with the value before the comparison with the value from the file is made. By default, the mask shall be interpreted as an unsigned decimal number. With a leading 0x or 0X, the mask shall be interpreted as an unsigned hexadecimal number; otherwise, with a leading 0, the mask shall be interpreted as an unsigned octal number.

The strings **byte**, **short**, **long**, and **string** shall also be supported as type fields, being interpreted as `dC`, `dS`, `dL`, and `s`, respectively.

value The *value* to be compared with the value from the file.

If the specifier from the type field is `s` or **string**, then interpret the value as a string. Otherwise, interpret it as a number. If the value is a string, then the test shall succeed only when a string value exactly matches the bytes from the file.

If the *value* is a string, it can contain the following sequences:

character The backslash-escape sequences as specified in the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`). The results of using any other character, other than an octal digit, following the backslash are unspecified.

octal	Octal sequences that can be used to represent characters with specific coded values. An octal sequence shall consist of a backslash followed by the longest sequence of one, two, or three octal-digit characters (01234567). If the size of a byte on the system is greater than 9 bits, the valid escape sequence used to represent a byte is implementation-defined.

By default, any value that is not a string shall be interpreted as a signed decimal number. Any such value, with a leading 0x or 0X, shall be interpreted as an unsigned hexadecimal number; otherwise, with a leading zero, the value shall be interpreted as an unsigned octal number.

If the value is not a string, it can be preceded by a character indicating the comparison to be performed. Permissible characters and the comparisons they specify are as follows:

=	The test shall succeed if the value from the file equals the *value* field.
<	The test shall succeed if the value from the file is less than the *value* field.
>	The test shall succeed if the value from the file is greater than the *value* field.
&	The test shall succeed if all of the bits in the *value* field are set in the value from the file.
^	The test shall succeed if at least one of the bits in the *value* field is not set in the value from the file.
x	The test shall succeed if the file is large enough to contain a value of the type specified starting at the offset specified.

message	The *message* to be printed if the test succeeds. The *message* shall be interpreted using the notation for the *printf* formatting specification; see *printf*. If the *value* field was a string, then the value from the file shall be the argument for the *printf* formatting specification; otherwise, the value from the file shall be the argument.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *file* utility can only be required to guess at many of the file types because only exhaustive testing can determine some types with certainty. For example, binary data on some implementations might match the initial segment of an executable or a *tar* archive. |

Note that the table indicates that the output contains the stated string. Systems may add text before or after the string. For executables, as an example, the machine architecture and various facts about how the file was link-edited may be included.

EXAMPLES

Determine whether an argument is a binary executable file:

```
file "$1" | grep −Fq executable &&
    printf "%s is executable.\n" "$1"
```

RATIONALE

The −**f** option was omitted because the same effect can (and should) be obtained using the *xargs* utility.

Historical versions of the *file* utility attempt to identify the following types of files: symbolic link, directory, character special, block special, socket, *tar* archive, *cpio* archive, SCCS archive, archive library, empty, *compress* output, *pack* output, binary data, C source, FORTRAN source, assembler source, *nroff*/*troff*/*eqn*/*tbl* source *troff* output, shell script, C shell script, English text, ASCII text, various executables, APL workspace, compiled terminfo entries, and CURSES screen images. Only those types that are reasonably well specified in POSIX or are directly related to POSIX utilities are listed in the table.

Implementations that support symbolic links are encouraged to use the string `"symbolic link"` to identify them.

Historical systems have used a ‘‘magic file’’ named **/etc/magic** to help identify file types. Because it is generally useful for users and scripts to be able to identify special file types, the −**m** flag and a portable format for user-created magic files has been specified. No requirement is made that an implementation of *file* use this method of identifying files, only that users be permitted to add their own classifying tests.

In addition, three options have been added to historical practice. The −**d** flag has been added to permit users to cause their tests to follow any default system tests. The −**i** flag has been added to permit users to test portably for regular files in shell scripts. The −**M** flag has been added to permit users to ignore any default system tests.

The historical −**c** option was omitted as not particularly useful to users or portable shell scripts. In addition, a reasonable implementation of the *file* utility would report any errors found each time the magic file is read.

The historical format of the magic file was the same as that specified by the Rationale in the previous version of IEEE Std 1003.1-200x for the *offset*, *value*, and *message* fields; however, it used less precise type fields than the format specified by the current normative text. The new type field values are a superset of the historical ones.

The following is an example magic file:

```
0  short     070707              cpio archive
0  short     0143561             Byte-swapped cpio archive
0  string    070707              ASCII cpio archive
0  long      0177555             Very old archive
0  short     0177545             Old archive
0  short     017437              Old packed data
0  string    \037\036            Packed data
0  string    \377\037            Compacted data
0  string    \037\235            Compressed data
>2 byte&0x80 >0                  Block compressed
>2 byte&0x1f x                   %d bits
0  string    \032\001            Compiled Terminfo Entry
0  short     0433                Curses screen image
0  short     0434                Curses screen image
```

```
0  string    <ar>                 System V Release 1 archive
0  string    !<arch>\n__.SYMDEF  Archive random library
0  string    !<arch>              Archive
0  string    ARF_BEGARF           PHIGS clear text archive
0  long      0x137A2950           Scalable OpenFont binary
0  long      0x137A2951           Encrypted scalable OpenFont binary
```

The use of a basic integer data type is intended to allow the implementation to choose a word size commonly used by applications on that architecture.

FUTURE DIRECTIONS

None.

SEE ALSO

ls

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

Options and an EXTENDED DESCRIPTION are added as specified in the IEEE P1003.2b draft standard.

IEEE PASC Interpretations 1003.2 #192 and #178 are applied.

NAME

find — find files

SYNOPSIS

`find [−H | −L] path ... [operand_expression ...]`

DESCRIPTION

The *find* utility shall recursively descend the directory hierarchy from each file specified by *path*, evaluating a Boolean expression composed of the primaries described in the OPERANDS section for each file encountered.

The *find* utility shall be able to descend to arbitrary depths in a file hierarchy and shall not fail due to path length limitations (unless a *path* operand specified by the application exceeds {PATH_MAX} requirements).

The *find* utility shall detect infinite loops; that is, entering a previously visited directory that is an ancestor of the last file encountered. When it detects an infinite loop, *find* shall write a diagnostic message to standard error and shall either recover its position in the hierarchy or terminate.

OPTIONS

The *find* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−H Cause the file information and file type evaluated for each symbolic link encountered on the command line to be those of the file referenced by the link, and not the link itself. If the referenced file does not exist, the file information and type shall be for the link itself. File information for all symbolic links not on the command line shall be that of the link itself.

−L Cause the file information and file type evaluated for each symbolic link to be those of the file referenced by the link, and not the link itself.

Specifying more than one of the mutually-exclusive options −**H** and −**L** shall not be considered an error. The last option specified shall determine the behavior of the utility.

OPERANDS

The following operands shall be supported:

The *path* operand is a pathname of a starting point in the directory hierarchy.

The first argument that starts with a `'−'`, or is a `'!'` or a `'('`, and all subsequent arguments shall be interpreted as an *expression* made up of the following primaries and operators. In the descriptions, wherever *n* is used as a primary argument, it shall be interpreted as a decimal integer optionally preceded by a plus (`'+'`) or minus (`'−'`) sign, as follows:

+*n* More than *n*.

n Exactly *n*.

−*n* Less than *n*.

The following primaries shall be supported:

−**name** *pattern*
The primary shall evaluate as true if the basename of the filename being examined matches *pattern* using the pattern matching notation described in Section 2.13 (on page 2264).

−nouser The primary shall evaluate as true if the file belongs to a user ID for which the *getpwuid*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x (or equivalent) returns NULL.

−nogroup The primary shall evaluate as true if the file belongs to a group ID for which the *getgrgid*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x (or equivalent) returns NULL.

−xdev The primary always shall evaluate as true; it shall cause *find* not to continue descending past directories that have a different device ID (*st_dev*, see the *stat*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x). If any −**xdev** primary is specified, it shall apply to the entire expression even if the −**xdev** primary would not normally be evaluated.

−prune The primary always shall evaluate as true; it shall cause *find* not to descend the current pathname if it is a directory. If the −**depth** primary is specified, the −**prune** primary shall have no effect.

−perm [−]*mode*

The *mode* argument is used to represent file mode bits. It shall be identical in format to the *symbolic_mode* operand described in *chmod* (on page 2438), and shall be interpreted as follows. To start, a template shall be assumed with all file mode bits cleared. An *op* symbol of `'+'` shall set the appropriate mode bits in the template; `'−'` shall clear the appropriate bits; `'='` shall set the appropriate mode bits, without regard to the contents of process' file mode creation mask. The *op* symbol of `'−'` cannot be the first character of *mode*; this avoids ambiguity with the optional leading hyphen. Since the initial mode is all bits off, there are not any symbolic modes that need to use `'−'` as the first character.

If the hyphen is omitted, the primary shall evaluate as true when the file permission bits exactly match the value of the resulting template.

Otherwise, if *mode* is prefixed by a hyphen, the primary shall evaluate as true if at least all the bits in the resulting template are set in the file permission bits.

−perm [−]*onum*

If the hyphen is omitted, the primary shall evaluate as true when the file permission bits exactly match the value of the octal number *onum* and only the bits corresponding to the octal mask 07777 shall be compared. (See the description of the octal *mode* in *chmod* (on page 2438).) Otherwise, if *onum* is prefixed by a hyphen, the primary shall evaluate as true if at least all of the bits specified in *onum* that are also set in the octal mask 07777 are set.

−type *c* The primary shall evaluate as true if the type of the file is *c*, where *c* is `'b'`, `'c'`, `'d'`, `'l'`, `'p'`, `'f'`, or `'s'` for block special file, character special file, directory, symbolic link, FIFO, regular file, or socket, respectively.

−links *n* The primary shall evaluate as true if the file has *n* links.

−user *uname* The primary shall evaluate as true if the file belongs to the user *uname*. If *uname* is a decimal integer and the *getpwnam*( ) (or equivalent) function does not return a valid user name, *uname* shall be interpreted as a user ID.

−group *gname*

The primary shall evaluate as true if the file belongs to the group *gname*. If *gname* is a decimal integer and the *getgrnam*( ) (or equivalent) function does not return a valid group name, *gname* shall be interpreted as a group ID.

−size *n*[**c**] The primary shall evaluate as true if the file size in bytes, divided by 512 and rounded up to the next integer, is *n*. If *n* is followed by the character `'c'`, the size shall be in bytes.

−atime *n* The primary shall evaluate as true if the file access time subtracted from the initialization time, divided by 86 400 (with any remainder discarded), is *n*.

−ctime *n* The primary shall evaluate as true if the time of last change of file status information subtracted from the initialization time, divided by 86 400 (with any remainder discarded), is *n*.

−mtime *n* The primary shall evaluate as true if the file modification time subtracted from the initialization time, divided by 86 400 (with any remainder discarded), is *n*.

−exec *utility_name* **[***argument* ...**] ;**
−exec *utility_name* **[***argument* ...**] ;** {} +

The end of the primary expression shall be punctuated by a semicolon or by a plus sign. Only a plus sign that follows an argument containing the two characters `"{}"` shall punctuate the end of the primary expression. Other uses of the plus sign shall not be treated as special.

If the primary expression is punctuated by a semicolon, the utility *utility_name* shall be invoked once for each pathname and the primary shall evaluate as true if the utility returns a zero value as exit status. A *utility_name* or *argument* containing only the two characters `"{}"` shall be replaced by the current pathname.

If the primary expression is punctuated by a plus sign, the primary shall always evaluate as true, and the pathnames for which the primary is evaluated shall be aggregated into sets. The utility *utility_name* shall be invoked once for each set of aggregated pathnames. Each invocation shall begin after the last pathname in the set is aggregated, and shall be completed before the *find* utility exits and before the first pathname in the next set (if any) is aggregated for this primary, but it is otherwise unspecified whether the invocation occurs before, during, or after the evaluations of other primaries. If any invocation returns a non-zero value as exit status, the *find* utility shall return a non-zero exit status. An argument containing only the two characters `"{}"` shall be replaced by the set of aggregated pathnames, with each pathname passed as a separate argument to the invoked utility in the same order that it was aggregated. The size of any set of two or more pathnames shall be limited such that execution of the utility does not cause the system's {ARG_MAX} limit to be exceeded. If more than one argument containing only the two characters `"{}"` is present, the behavior is unspecified.

If a *utility_name* or *argument* string contains the two characters `"{}"`, but not just the two characters `"{}"`, it is implementation-defined whether *find* replaces those two characters or uses the string without change. The current directory for the invocation of *utility_name* shall be the same as the current directory when the *find* utility was started. If the *utility_name* names any of the special built-in utilities (see Section 2.14 (on page 2266)), the results are undefined.

−ok *utility_name* **[***argument* ...**] ;**

The **−ok** primary shall be equivalent to **−exec**, except that the use of a plus sign to punctuate the end of the primary expression need not be supported, and *find* shall request affirmation of the invocation of *utility_name* using the current file as an argument by writing to standard error as described in the STDERR section. If the response on standard input is affirmative, the utility shall be invoked. Otherwise, the command shall not be invoked and the value of the **−ok** operand shall be false.

–print The primary always shall evaluate as true; it shall cause the current pathname to be written to standard output.

–newer *file* The primary shall evaluate as true if the modification time of the current file is more recent than the modification time of the file named by the pathname *file*.

–depth The primary shall always evaluate as true; it shall cause descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. If a **–depth** primary is not specified, all entries in a directory shall be acted on after the directory itself. If any **–depth** primary is specified, it shall apply to the entire expression even if the **–depth** primary would not normally be evaluated.

The primaries can be combined using the following operators (in order of decreasing precedence):

(*expression*) True if *expression* is true.

! *expression* Negation of a primary; the unary NOT operator.

expression **[–a]** *expression*

Conjunction of primaries; the AND operator is implied by the juxtaposition of two primaries or made explicit by the optional **–a** operator. The second expression shall not be evaluated if the first expression is false.

expression **–o** *expression*

Alternation of primaries; the OR operator. The second expression shall not be evaluated if the first expression is true.

If no *expression* is present, **–print** shall be used as the expression. Otherwise, if the given expression does not contain any of the primaries **–exec**, **–ok**, or **–print**, the given expression shall be effectively replaced by:

```
( given_expression ) -print
```

The **–user**, **–group**, and **–newer** primaries each shall evaluate their respective arguments only once.

STDIN

If the **–ok** primary is used, the response shall be read from the standard input. An entire line shall be read as the response. Otherwise, the standard input shall not be used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *find*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE

Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements used in the pattern matching notation for the **–n** option and in the extended regular expression defined for the **yesexpr** locale

	keyword in the *LC_MESSAGES* category.
LC_CTYPE	This variable determines the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments), the behavior of character classes within the pattern matching notation used for the −**n** option, and the behavior of character classes within regular expressions used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_MESSAGES	Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the location of the *utility_name* for the −**exec** and −**ok** primaries, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The −**print** primary shall cause the current pathnames to be written to standard output. The format shall be:

```
"%s\n", <path>
```

STDERR

The −**ok** primary shall write a prompt to standard error containing at least the *utility_name* to be invoked and the current pathname. In the POSIX locale, the last non-<blank> in the prompt shall be `'?'`. The exact format used is unspecified.

Otherwise, the standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All *path* operands were traversed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

When used in operands, pattern matching notation, semicolons, opening parentheses, and closing parentheses are special to the shell and must be quoted (see Section 2.2 (on page 2232)).

The bit that is traditionally used for sticky (historically 01000) is specified in the −**perm** primary using the octal number argument form. Since this bit is not defined by this volume of IEEE Std 1003.1-200x, applications must not assume that it actually refers to the traditional sticky bit.

EXAMPLES

1. The following commands are equivalent:

   ```
   find .
   find . −print
   ```

   They both write out the entire directory hierarchy from the current directory.

2. The following command:

   ```
   find / \( −name tmp −o −name ’*.xx’ \) −atime +7 −exec rm {} \;
   ```

   removes all files named **tmp** or ending in **.xx** that have not been accessed for seven or more 24-hour periods.

3. The following command:

   ```
   find . −perm −o+w,+s
   ```

   prints (−**print** is assumed) the names of all files in or below the current directory, with all of the file permission bits S_ISUID, S_ISGID, and S_IWOTH set.

4. The following command:

   ```
   find . −name SCCS −prune −o −print
   ```

   recursively prints pathnames of all files in the current directory and below, but skips directories named SCCS and files in them.

5. The following command:

   ```
   find . −print −name SCCS −prune
   ```

   behaves as in the previous example, but prints the names of the SCCS directories.

6. The following command is roughly equivalent to the −**nt** extension to *test*:

   ```
   if [ −n "$(find file1 −prune −newer file2)" ]; then
       printf %s\\n "file1 is newer than file2"
   fi
   ```

7. The descriptions of −**atime**, −**ctime**, and −**mtime** use the terminology *n* ‘‘86 400 second periods (days)’’. For example, a file accessed at 23:59 is selected by:

   ```
   find . −atime −1 −print
   ```

   at 00:01 the next day (less than 24 hours later, not more than one day ago); the midnight boundary between days has no effect on the 24-hour calculation.

RATIONALE

The −**a** operator was retained as an optional operator for compatibility with historical shell scripts, even though it is redundant with expression concatenation.

The descriptions of the `'−'` modifier on the *mode* and *onum* arguments to the −**perm** primary agree with historical practice on BSD and System V implementations. System V and BSD documentation both describe it in terms of checking additional bits; in fact, it uses the same bits, but checks for having at least all of the matching bits set instead of having exactly the matching bits set.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified because:

- Implementations may desire more descriptive prompts than those used on historical implementations.
- Since the historical prompt strings do not terminate with <newline>s, there is no portable way for another program to interact with the prompts of this utility via pipes.

Therefore, an application using this prompting option relies on the system to provide the most suitable dialog directly with the user, based on the general guidelines specified.

The −**name** *file* operand was changed to use the shell pattern matching notation so that *find* is consistent with other utilities using pattern matching.

The −**size** operand refers to the size of a file, rather than the number of blocks it may occupy in the file system. The intent is that the *st_size* field defined in the System Interfaces volume of IEEE Std 1003.1-200x should be used, not the *st_blocks* found in historical implementations. There are at least two reasons for this:

1. In both System V and BSD, *find* only uses *st_size* in size calculations for the operands specified by this volume of IEEE Std 1003.1-200x. (BSD uses *st_blocks* only when processing the −**ls** primary.)
2. Users usually think of file size in terms of bytes, which is also the unit used by the *ls* utility for the output from the −**l** option. (In both System V and BSD, *ls* uses *st_size* for the −**l** option size field and uses *st_blocks* for the *ls* −**s** calculations. This volume of IEEE Std 1003.1-200x does not specify *ls* −**s**.)

The descriptions of −**atime**, −**ctime**, and −**mtime** were changed from the SVID description of *n* ‘‘days’’ to ‘‘24-hour periods’’. The description is also different in terms of the exact timeframe for the *n* case (*versus* the +*n* or −*n*), but it matches all known historical implementations. It refers to one 86 400 second period in the past, not any time from the beginning of that period to the current time. For example, −**atime** 3 is true if the file was accessed any time in the period from 72 hours to 48 hours ago.

Historical implementations do not modify `"{}"` when it appears as a substring of an −**exec** or −**ok** *utility_name* or argument string. There have been numerous user requests for this extension, so this volume of IEEE Std 1003.1-200x allows the desired behavior. At least one recent implementation does support this feature, but encountered several problems in managing memory allocation and dealing with multiple occurrences of `"{}"` in a string while it was being developed, so it is not yet required behavior.

Assuming the presence of −**print** was added to correct a historical pitfall that plagues novice users, it is entirely upward-compatible from the historical System V *find* utility. In its simplest form (*find directory*), it could be confused with the historical BSD fast *find*. The BSD developers agreed that adding −**print** as a default expression was the correct decision and have added the fast *find* functionality within a new utility called *locate*.

Historically, the −**L** option was implemented using the primary −**follow**. The −**H** and −**L** options were added for two reasons. First, they offer a finer granularity of control and consistency with other programs that walk file hierarchies. Second, the −**follow** primary always evaluated to true.

As they were historically really global variables that took effect before the traversal began, some valid expressions had unexpected results. An example is the expression −**print** −**o** −**follow**. Because −**print** always evaluates to true, the standard order of evaluation implies that −**follow** would never be evaluated. This was never the case. Historical practice for the −**follow** primary, however, is not consistent. Some implementations always follow symbolic links on the command line whether −**follow** is specified or not. Others follow symbolic links on the command line only if −**follow** is specified. Both behaviors are provided by the −**H** and −**L** options, but scripts using the current −**follow** primary would be broken if the −**follow** option is specified to work either way.

Since the −**L** option resolves all symbolic links and the −**type** *l* primary is true for symbolic links that still exist after symbolic links have been resolved, the command:

```
find −L . −type l
```

prints a list of symbolic links reachable from the current directory that do not resolve to accessible files.

A feature of SVR4's *find* utility was the −**exec** primary's **+** terminator. This allowed filenames containing special characters (especially <newline>s) to be grouped together without the problems that occur if such filenames are piped to *xargs*. Other implementations have added other ways to get around this problem, notably a −**print0** primary that wrote filenames with a null byte terminator. This was considered here, but not adopted. Using a null terminator meant that any utility that was going to process *find*'s −**print0** output had to add a new option to parse the null terminators it would now be reading.

The `"−exec ... {} +"` syntax adopted was a result of IEEE PASC Interpretation 1003.2 #210. It should be noted that this is an incompatible change to the ISO/IEC 9899:1999 standard. For example, the following command prints all files with a `'−'` after their name if they are regular files, and a `'+'` otherwise:

```
find / −type f −exec echo {} − ';' −o −exec echo {} + ';'
```

The change invalidates usage like this. Even though the previous standard stated that this usage would work, in practice many did not support it and the standard developers felt it better to now state that this was not allowable.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, *pax*, *sh*, *test*, the System Interfaces volume of IEEE Std 1003.1-200x, *stat*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The −**perm [**−**]***onum* primary is supported.

The *find* utility is aligned with the IEEE P1003.2b draft standard, to include processing of symbolic links and changes to the description of the **atime**, **ctime**, and **mtime** operands.

IEEE PASC Interpretation 1003.2 #210 is applied, extending the −**exec** operand.

NAME

fold — filter for folding lines

SYNOPSIS

```
fold [−bs][−w width][file...]
```

DESCRIPTION

The *fold* utility is a filter that shall fold lines from its input files, breaking the lines to have a maximum of *width* column positions (or bytes, if the **−b** option is specified). Lines shall be broken by the insertion of a <newline> such that each output line (referred to later in this section as a *segment*) is the maximum width possible that does not exceed the specified number of column positions (or bytes). A line shall not be broken in the middle of a character. The behavior is undefined if *width* is less than the number of columns any single character in the input would occupy.

If the <carriage-return>s, <backspace>s, or <tab>s are encountered in the input, and the **−b** option is not specified, they shall be treated specially:

<backspace>	The current count of line width shall be decremented by one, although the count never shall become negative. The *fold* utility shall not insert a <newline> immediately before or after any <backspace>.
<carriage-return>	The current count of line width shall be set to zero. The *fold* utility shall not insert a <newline> immediately before or after any <carriage-return>.
<tab>	Each <tab> encountered shall advance the column position pointer to the next tab stop. Tab stops shall be at each column position *n* such that *n* modulo 8 equals 1.

OPTIONS

The *fold* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−b	Count *width* in bytes rather than column positions.
−s	If a segment of a line contains a <blank> within the first *width* column positions (or bytes), break the line after the last such <blank> meeting the width constraints. If there is no <blank> meeting the requirements, the **−s** option shall have no effect for that output segment of the input line.
−w *width*	Specify the maximum line length, in column positions (or bytes if **−b** is specified). The results are unspecified if *width* is not a positive decimal number. The default value shall be 80.

OPERANDS

The following operand shall be supported:

file	A pathname of a text file to be folded. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

If the −**b** option is specified, the input files shall be text files except that the lines are not limited to {LINE_MAX} bytes in length. If the −**b** option is not specified, the input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *fold*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), and for the determination of the width in column positions each character would occupy on a constant-width font output device.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be a file containing a sequence of characters whose order shall be preserved from the input files, possibly with inserted <newline>s.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were processed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *cut* and *fold* utilities can be used to create text files out of files with arbitrary line lengths. The *cut* utility should be used when the number of lines (or records) needs to remain constant. The *fold* utility should be used when the contents of long lines need to be kept contiguous.

The *fold* utility is frequently used to send text files to printers that truncate, rather than fold, lines wider than the printer is able to print (usually 80 or 132 column positions).

EXAMPLES

An example invocation that submits a file of possibly long lines to the printer (under the assumption that the user knows the line width of the printer to be assigned by *lp*):

```
fold −w 132 bigfile | lp
```

RATIONALE

Although terminal input in canonical processing mode requires the erase character (frequently set to <backspace>) to erase the previous character (not byte or column position), terminal output is not buffered and is extremely difficult, if not impossible, to parse correctly; the interpretation depends entirely on the physical device that actually displays/prints/stores the output. In all known internationalized implementations, the utilities producing output for mixed column-width output assume that a <backspace> backs up one column position and outputs enough <backspace>s to return to the start of the character when <backspace> is used to provide local line motions to support underlining and emboldening operations. Since *fold* without the −**b** option is dealing with these same constraints, <backspace> is always treated as backing up one column position rather than backing up one character.

Historical versions of the *fold* utility assumed 1 byte was one character and occupied one column position when written out. This is no longer always true. Since the most common usage of *fold* is believed to be folding long lines for output to limited-length output devices, this capability was preserved as the default case. The −**b** option was added so that applications could *fold* files with arbitrary length lines into text files that could then be processed by the standard utilities. Note that although the width for the −**b** option is in bytes, a line is never split in the middle of a character. (It is unspecified what happens if a width is specified that is too small to hold a single character found in the input followed by a <newline>.)

The tab stops are hardcoded to be every eighth column to meet historical practice. No new method of specifying other tab stops was invented.

FUTURE DIRECTIONS

None.

SEE ALSO

cut

CHANGE HISTORY

First released in Issue 4.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

fort77 — FORTRAN compiler (**FORTRAN**)

SYNOPSIS

FD

```
fort77 [-c][-g][-L directory]... [-O optlevel][-o outfile][-s][-w]
    operand...
```

DESCRIPTION

The *fort77* utility is the interface to the FORTRAN compilation system; it shall accept the full FORTRAN-77 language defined by the ANSI X3.9-1978 standard. The system conceptually consists of a compiler and link editor. The files referenced by *operand*s are compiled and linked to produce an executable file. It is unspecified whether the linking occurs entirely within the operation of *fort77*; some implementations may produce objects that are not fully resolved until the file is executed.

If the **−c** option is present, for all pathname operands of the form *file*.**f**, the files:

`$(basename pathname.f).o`

shall be created or overwritten as the result of successful compilation. If the **−c** option is not specified, it is unspecified whether such **.o** files are created or deleted for the *file*.**f** operands.

If there are no options that prevent link editing (such as **−c**) and all operands compile and link without error, the resulting executable file shall be written into the file named by the **−o** option (if present) or to the file **a.out**. The executable file shall be created as specified in the System Interfaces volume of IEEE Std 1003.1-200x, except that the file permissions shall be set to:

S_IRWXO | S_IRWXG | S_IRWXU

and that the bits specified by the *umask* of the process shall be cleared.

OPTIONS

The *fort77* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that:

- The **−l** *library* operands have the format of options, but their position within a list of operands affects the order in which libraries are searched.
- The order of specifying the multiple **−L** options is significant.
- Conforming applications shall specify each option separately; that is, grouping option letters (for example, **−cg**) need not be recognized by all implementations.

The following options shall be supported:

−c	Suppress the link-edit phase of the compilation, and do not remove any object files that are produced.
−g	Produce symbolic information in the object or executable files; the nature of this information is unspecified, and may be modified by implementation-defined interactions with other options.
−s	Produce object or executable files, or both, from which symbolic and other information not required for proper execution using the *exec* family of functions defined in the System Interfaces volume of IEEE Std 1003.1-200x has been removed (stripped). If both **−g** and **−s** options are present, the action taken is unspecified.
−o *outfile*	Use the pathname *outfile*, instead of the default **a.out**, for the executable file produced. If the **−o** option is present with **−c**, the result is unspecified.

–L *directory* Change the algorithm of searching for the libraries named in **–l** operands to look in the directory named by the *directory* pathname before looking in the usual places. Directories named in **–L** options shall be searched in the specified order. At least ten instances of this option shall be supported in a single *fort77* command invocation. If a directory specified by a **–L** option contains a file named **libf.a**, the results are unspecified.

–O *optlevel* Specify the level of code optimization. If the *optlevel* option-argument is the digit `'0'`, all special code optimizations shall be disabled. If it is the digit `'1'`, the nature of the optimization is unspecified. If the **–O** option is omitted, the nature of the system's default optimization is unspecified. It is unspecified whether code generated in the presence of the **–O** 0 option is the same as that generated when **–O** is omitted. Other *optlevel* values may be supported.

–w Suppress warnings.

Multiple instances of **–L** options can be specified.

OPERANDS

An *operand* is either in the form of a pathname or the form **–l** *library*. At least one operand of the pathname form shall be specified. The following operands shall be supported:

file.**f** The pathname of a FORTRAN source file to be compiled and optionally passed to the link editor. The filename operand shall be of this form if the **–c** option is used.

file.**a** A library of object files typically produced by *ar*, and passed directly to the link editor. Implementations may recognize implementation-defined suffixes other than **.a** as denoting object file libraries.

file.**o** An object file produced by *fort77* **–c** and passed directly to the link editor. Implementations may recognize implementation-defined suffixes other than **.o** as denoting object files.

The processing of other files is implementation-defined.

–l *library* (The letter ell.) Search the library named:

`lib`*`library`*`.a`

A library is searched when its name is encountered, so the placement of a **–l** operand is significant. Several standard libraries can be specified in this manner, as described in the EXTENDED DESCRIPTION section. Implementations may recognize implementation-defined suffixes other than **.a** as denoting libraries.

STDIN

Not used.

INPUT FILES

The input file shall be one of the following: a text file containing FORTRAN source code; an object file in the format produced by *fort77* **–c**; or a library of object files, in the format produced by archiving zero or more object files, using *ar*. Implementations may supply additional utilities that produce files in these formats. Additional input files are implementation-defined.

A <tab> encountered within the first six characters on a line of source code shall cause the compiler to interpret the following character as if it were the seventh character on the line (that is, in column 7).

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *fort77*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
TMPDIR	Determine the pathname that should override the default directory for temporary files, if any.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages. If more than one *file* operand ending in **.f** (or possibly other unspecified suffixes) is given, for each such file: |

```
"%s:\n", <file>
```

may be written to allow identification of the diagnostic message with the appropriate input file.

This utility may produce warning messages about certain conditions that do not warrant returning an error (non-zero) exit value.

OUTPUT FILES

Object files, listing files and executable files shall be produced in unspecified formats.

EXTENDED DESCRIPTION

Standard Libraries

The *fort77* utility shall recognize the following **–l** operand for the standard library:

–l f	This library contains all functions referenced in the ANSI X3.9-1978 standard. This operand shall not be required to be present to cause a search of this library.

In the absence of options that inhibit invocation of the link editor, such as **–c**, the *fort77* utility shall cause the equivalent of a **–l f** operand to be passed to the link editor as the last **–l** operand, causing it to be searched after all other object files and libraries are loaded.

It is unspecified whether the library **libf.a** exists as a regular file. The implementation may accept as **–l** operands names of objects that do not exist as regular files.

External Symbols

The FORTRAN compiler and link editor shall support the significance of external symbols up to a length of at least 31 bytes; case folding is permitted. The action taken upon encountering symbols exceeding the implementation-defined maximum symbol length is unspecified.

The compiler and link editor shall support a minimum of 511 external symbols per source or object file, and a minimum of 4 095 external symbols total. A diagnostic message is written to standard output if the implementation-defined limit is exceeded; other actions are unspecified.

EXIT STATUS

The following exit values shall be returned:

0	Successful compilation or link edit.
>0	An error occurred.

CONSEQUENCES OF ERRORS

When *fort77* encounters a compilation error, it shall write a diagnostic to standard error and continue to compile other source code operands. It shall return a non-zero exit status, but it is implementation-defined whether an object module is created. If the link edit is unsuccessful, a diagnostic message shall be written to standard error, and *fort77* shall exit with a non-zero status.

APPLICATION USAGE

None.

EXAMPLES

The following usage example compiles **xyz.f** and creates the executable file **foo**:

```
fort77 −o foo xyz.f
```

The following example compiles **xyz.f** and creates the object file **xyz.o**:

```
fort77 −c xyz.f
```

The following example compiles **xyz.f** and creates the executable file **a.out**:

```
fort77 xyz.f
```

The following example compiles **xyz.f**, links it with **b.o**, and creates the executable **a.out**:

```
fort77 xyz.f b.o
```

RATIONALE

The name of this utility was chosen as *fort77* to parallel the renaming of the C compiler. The name *f77* was not chosen to avoid problems with historical implementations. The ANSI X3.9-1978 standard was selected as a normative reference because the ISO / IEC version of FORTRAN-77 has been superseded by the ISO / IEC 1539: 1990 standard (Fortran-90).

The file inclusion and symbol definition **#define** mechanisms used by the *c99* utility were not included in this volume of IEEE Std 1003.1-200x—even though they are commonly implemented—since there is no requirement that the FORTRAN compiler use the C preprocessor.

The −**onetrip** option was not included in this volume of IEEE Std 1003.1-200x, even though many historical compilers support it, because it is derived from FORTRAN-66; it is an anachronism that should not be perpetuated.

Some implementations produce compilation listings. This aspect of FORTRAN has been left unspecified because there was controversy concerning the various methods proposed for implementing it: a −**V** option overlapped with historical vendor practice and a naming convention of creating files with **.l** suffixes collided with historical *lex* file naming practice.

There is no **–I** option in this version of this volume of IEEE Std 1003.1-200x to specify a directory for file inclusion. An INCLUDE directive has been a part of the Fortran-90 discussions, but an interface supporting that standard is not in the current scope.

It is noted that many FORTRAN compilers produce an object module even when compilation errors occur; during a subsequent compilation, the compiler may patch the object module rather than recompiling all the code. Consequently, it is left to the implementor whether or not an object file is created.

A reference to MIL-STD-1753 was removed from an early proposal in response to a request from the POSIX FORTRAN-binding standard developers. It was not the intention of the standard developers to require certification of the FORTRAN compiler, and IEEE Std 1003.9-1992 does not specify the military standard or any special preprocessing requirements. Furthermore, use of that document would have been inappropriate for an international standard.

The specification of optimization has been subject to changes through early proposals. At one time, **–O** and **–N** were Booleans: optimize and do not optimize (with an unspecified default). Some historical practice lead this to be changed to:

–O 0	No optimization.
–O 1	Some level of optimization.
–O *n*	Other, unspecified levels of optimization.

It is not always clear whether ‘‘good code generation’’ is the same thing as optimization. Simple optimizations of local actions do not usually affect the semantics of a program. The **–O** 0 option has been included to accommodate the very particular nature of scientific calculations in a highly optimized environment; compilers make errors. Some degree of optimization is expected, even if it is not documented here, and the ability to shut it off completely could be important when porting an application. An implementation may treat **–O** 0 as ‘‘do less than normal’’ if it wishes, but this is only meaningful if any of the operations it performs can affect the semantics of a program. It is highly dependent on the implementation whether doing less than normal is logical. It is not the intent of the **–O** 0 option to ask for inefficient code generation, but rather to assure that any semantically visible optimization is suppressed.

The specification of standard library access is consistent with the C compiler specification. Implementations are not required to have **/usr/lib/libf.a**, as many historical implementations do, but if not they are required to recognize **f** as a token.

External symbol size limits are in normative text; conforming applications need to know these limits. However, the minimum maximum symbol length should be taken as a constraint on a conforming application, not on an implementation, and consequently the action taken for a symbol exceeding the limit is unspecified. The minimum size for the external symbol table was added for similar reasons.

The CONSEQUENCES OF ERRORS section clearly specifies the behavior of the compiler when compilation or link-edit errors occur. The behavior of several historical implementations was examined, and the choice was made to be silent on the status of the executable, or **a.out**, file in the face of compiler or linker errors. If a linker writes the executable file, then links it on disk with *lseek*( )s and *write*( )s, the partially linked executable file can be left on disk and its execute bits turned off if the link edit fails. However, if the linker links the image in memory before writing the file to disk, it need not touch the executable file (if it already exists) because the link edit fails. Since both approaches are historical practice, a conforming application shall rely on the exit status of *fort77*, rather than on the existence or mode of the executable file.

The −**g** and −**s** options are not specified as mutually-exclusive. Historically these two options have been mutually-exclusive, but because both are so loosely specified, it seemed appropriate to leave their interaction unspecified.

The requirement that conforming applications specify compiler options separately is to reserve the multi-character option name space for vendor-specific compiler options, which are known to exist in many historical implementations. Implementations are not required to recognize, for example, −**gc** as if it were −**g** −**c**; nor are they forbidden from doing so. The SYNOPSIS shows all of the options separately to highlight this requirement on applications.

Echoing filenames to standard error is considered a diagnostic message because it would otherwise be difficult to associate an error message with the erring file. They are described with ‘‘may’’ to allow implementations to use other methods of identifying files and to parallel the description in *c99*.

FUTURE DIRECTIONS

A compilation system based on the ISO/IEC 1539:1990 standard (Fortran-90) may be considered for a future issue; it may have a different utility name from *fort77*.

SEE ALSO

ar, *asa*, *c99*, *umask*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the FORTRAN Development Utilities option.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

fuser — list process IDs of all processes that have one or more files open

SYNOPSIS

XSI
```
fuser [ -cfu ] file ...
```

DESCRIPTION

The *fuser* utility shall write to standard output the process IDs of processes running on the local system that have one or more named files open. For block special devices, all processes using any file on that device are listed.

The *fuser* utility shall write to standard error additional information about the named files indicating how the file is being used.

Any output for processes running on remote systems that have a named file open is unspecified.

A user may need appropriate privilege to invoke the *fuser* utility.

OPTIONS

The *fuser* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	The file is treated as a mount point and the utility shall report on any files open in the file system.
−f	The report shall be only for the named files.
−u	The user name, in parentheses, associated with each process ID written to standard output shall be written to standard error.

OPERANDS

The following operand shall be supported:

file	A pathname on which the file or file system is to be reported.

STDIN

Not used.

INPUT FILES

The user database.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *fuser*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *fuser* utility shall write the process ID for each process using each file given as an operand to standard output in the following format:

```
"%d", <process_id>
```

STDERR

The *fuser* utility shall write diagnostic messages to standard error.

The *fuser* utility also shall write the following to standard error:

- The pathname of each named file is written followed immediately by a colon.
- For each process ID written to standard output, the character `'c'` shall be written to standard error if the process is using the file as its current directory and the character `'r'` shall be written to standard error if the process is using the file as its root directory. Implementations may write other alphabetic characters to indicate other uses of files.
- When the **−u** option is specified, characters indicating the use of the file shall be followed immediately by the user name, in parentheses, corresponding to the process' real user ID. If the user name cannot be resolved from the process' real user ID, the process' real user ID shall be written instead of the user name.

When standard output and standard error are directed to the same file, the output shall be interleaved so that the filename appears at the start of each line, followed by the process ID and characters indicating the use of the file. Then, if the **−u** option is specified, the user name or user ID for each process using that file shall be written. |

A <newline> shall be written to standard error after the last output described above for each *file* operand.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

The command:

```
fuser –fu .
```

writes to standard output the process IDs of processes that are using the current directory and writes to standard error an indication of how those processes are using the directory and the user names associated with the processes that are using the current directory.

RATIONALE

The definition of the *fuser* utility follows existing practice.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 5.

NAME

gencat — generate a formatted message catalog

SYNOPSIS

XSI `gencat catfile msgfile...`

DESCRIPTION

The *gencat* utility shall merge the message text source files *msgfile* into a formatted message catalog *catfile*. The file *catfile* shall be created if it does not already exist. If *catfile* does exist, its messages shall be included in the new *catfile*. If set and message numbers collide, the new message text defined in *msgfile* shall replace the old message text currently contained in *catfile*.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

catfile	A pathname of the formatted message catalog. If '−' is specified, standard output shall be used. The format of the message catalog produced is unspecified.
msgfile	A pathname of a message text source file. If '−' is specified for an instance of *msgfile*, standard input shall be used. The format of message text source files is defined in the EXTENDED DESCRIPTION section.

STDIN

The standard input shall not be used unless a *msgfile* operand is specified as '−'.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *gencat*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall not be used unless the *catfile* operand is specified as '−'.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The content of a message text file shall be in the format defined as follows. Note that the fields of a message text source line are separated by a single <blank>. Any other <blank>s are considered as being part of the subsequent field.

$set *n comment*

This line specifies the set identifier of the following messages until the next **$set** or end-of-file appears. The *n* denotes the set identifier, which is defined as a number in the range [1, {NL_SETMAX}] (see the **<limits.h>** header defined in the System Interfaces volume of IEEE Std 1003.1-200x). The application shall ensure that set identifiers are presented in ascending order within a single source file, but need not be contiguous. Any string following the set identifier shall be treated as a comment. If no **$set** directive is specified in a message text source file, all messages shall be located in an implementation-defined default message set NL_SETD (see the **<nl_types.h>** header defined in the System Interfaces volume of IEEE Std 1003.1-200x).

$delset *n comment*

This line deletes message set *n* from an existing message catalog. The *n* denotes the set number [1, {NL_SETMAX}]. Any string following the set number shall be treated as a comment.

$ *comment* A line beginning with '$' followed by a <blank> shall be treated as a comment.

m message-text

The *m* denotes the message identifier, which is defined as a number in the range [1, {NL_MSGMAX}] (see the **<limits.h>** header defined in the System Interfaces volume of IEEE Std 1003.1-200x). The *message-text* shall be stored in the message catalog with the set identifier specified by the last **$set** directive, and with message identifier *m*. If the *message-text* is empty, and a <blank> field separator is present, an empty string shall be stored in the message catalog. If a message source line has a message number, but neither a field separator nor *message-text*, the existing message with that number (if any) shall be deleted from the catalog. The application shall ensure that message identifiers are in ascending order within a single set, but need not be contiguous. The application shall ensure that the length of *message-text* is in the range [0, {NL_TEXTMAX}] (see the **<limits.h>** header defined in the System Interfaces volume of IEEE Std 1003.1-200x).

$quote *n* This line specifies an optional quote character *c*, which can be used to surround *message-text* so that trailing spaces or null (empty) messages are visible in a message source line. By default, or if an empty **$quote** directive is supplied, no quoting of *message-text* shall be recognized.

Empty lines in a message text source file shall be ignored. The effects of lines starting with any character other than those defined above are implementation-defined.

Text strings can contain the special characters and escape sequences defined in the following table:

Description	Symbol	Sequence
<newline>	NL(LF)	`\n`
Horizontal tab	HT	`\t`
<vertical-tab>	VT	`\v`
<backspace>	BS	`\b`
<carriage-return>	CR	`\r`
<form-feed>	FF	`\f`
Backslash	`\`	`\\`
Bit pattern	`ddd`	`\ddd`

The escape sequence `"\ddd"` consists of backslash followed by one, two, or three octal digits, which shall be taken to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash shall be ignored.

Backslash (`'\'`) followed by a <newline> is also used to continue a string on the following line. Thus, the following two lines describe a single message string:

```
1 This line continues \
to the next line
```

which shall be equivalent to:

```
1 This line continues to the next line
```

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Message catalogs produced by *gencat* are binary encoded, meaning that their portability cannot be guaranteed between different types of machine. Thus, just as C programs need to be recompiled for each type of machine, so message catalogs must be recreated via *gencat*.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

iconv, the System Interfaces volume of IEEE Std 1003.1-200x, **<limits.h>**

CHANGE HISTORY

First released in Issue 3.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

get — get a version of an SCCS file (**DEVELOPMENT**)

SYNOPSIS

XSI
```
get [−begkmnlLpst][−c cutoff][−i list][−r SID][−x list] file...
```

DESCRIPTION

The *get* utility shall generate a text file from each named SCCS *file* according to the specifications given by its options.

The generated text shall normally be written into a file called the **g-file** whose name is derived from the SCCS filename by simply removing the leading `"s."`.

OPTIONS

The *get* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−r *SID* Indicate the SCCS Identification String (SID) of the version (delta) of an SCCS file to be retrieved. The table shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by *delta* if the **−e** option is also used), as a function of the SID specified.

−c *cutoff* Indicate the *cutoff* date-time, in the form:

YY[*MM*[*DD*[*HH*[*MM*[*SS*]]]]]

For the *YY* component, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive.

Note: It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time shall be included in the generated text file. Units omitted from the date-time default to their maximum possible values; for example, **−c** 7502 is equivalent to **−c** 750228235959.

Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows the user to specify a *cutoff* date in the form: **−c** "77/2/2 9:22:25".

−e Indicate that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*. The **−e** option used in a *get* for a particular version (SID) of the SCCS file shall prevent further *get* commands from editing on the same SID until *delta* is executed or the **j** (joint edit) flag is set in the SCCS file. Concurrent use of *get* **−e** for different SIDs is always allowed.

If the **g-file** generated by *get* with a **−e** option is accidentally ruined in the process of editing, it may be regenerated by re-executing the *get* command with the **−k** option in place of the **−e** option.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file shall be enforced when the **−e** option is used.

−b Use with the **−e** option to indicate that the new delta should have an SID in a new branch as shown in the table below. This option shall be ignored if the **b** flag is not present in the file or if the retrieved delta is not a leaf delta. (A leaf delta is one that has no successors on the SCCS file tree.)

Note: A branch delta may always be created from a non-leaf delta.

−i *list* Indicate a *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

```
<list> ::= <range> | <list> , <range>
<range> ::= SID | SID − SID
```

SID, the SCCS Identification of a delta, may be in any form shown in the ''SID Specified'' column of the table in the EXTENDED DESCRIPTION section, except that the result of supplying a partial SID is unspecified. A diagnostic message shall be written if the first SID in the range is not an ancestor of the second SID in the range.

−x *list* Indicate a *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **−i** option for the *list* format.

−k Suppress replacement of identification keywords (see below) in the retrieved text by their value. The **−k** option shall be implied by the **−e** option.

−l Write a delta summary into an **l-file**.

−L Write a delta summary to standard output. All informative output that normally is written to standard output shall be written to standard error instead, unless the **−s** option is used, in which case it shall be suppressed.

−p Write the text retrieved from the SCCS file to the standard output. No **g-file** shall be created. All informative output that normally goes to the standard output shall go to standard error instead, unless the **−s** option is used, in which case it shall disappear.

−s Suppress all informative output normally written to standard output. However, fatal error messages (which shall always be written to the standard error) shall remain unaffected.

−m Precede each text line retrieved from the SCCS file by the SID of the delta that inserted the text line in the SCCS file. The format shall be:

```
"%s\t%s", <SID>, <text line>
```

−n Precede each generated text line with the %**M**% identification keyword value (see below). The format shall be:

```
"%s\t%s", <%M% value>, <text line>
```

When both the **−m** and **−n** options are used, the <*text line*> shall be replaced by the **−m** option-generated format.

−g Suppress the actual retrieval of text from the SCCS file. It is primarily used to generate an **l-file**, or to verify the existence of a particular SID.

−t Use to access the most recently created (top) delta in a given release (for example, **−r 1**), or release and level (for example, **−r 1.2**).

OPERANDS

The following operands shall be supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, the *get* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files shall be silently ignored.

If exactly one *file* operand appears, and it is '−', the standard input shall be read; each line of the standard input is taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only if the *file* operand is specified as '−'. Each line of the text file shall be interpreted as an SCCS pathname.

INPUT FILES

The SCCS files shall be files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *get*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output (or standard error, if the −**p** option is used).
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
TZ	Determine the timezone in which the times and dates are written in the SCCS file are evaluated. If the *TZ* variable is unset or NULL, an unspecified system default timezone is used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

For each file processed, *get* shall write to standard output the SID being accessed and the number of lines retrieved from the SCCS file, in the following format:

```
"%s\n%d lines\n", <SID>, <number of lines>
```

If the −**e** option is used, the SID of the delta to be made shall appear after the SID accessed and before the number of lines generated, in the POSIX locale:

```
"%s\nnew delta %s\n%d lines\n", <SID accessed>,
    <SID to be made>, <number of lines>
```

If there is more than one named file or if a directory or standard input is named, each pathname shall be written before each of the lines shown in one of the preceding formats:

```
"\n%s:\n", <pathname>
```

If the −**L** option is used, a delta summary shall be written following the format specified below for **l-files**.

If the –**i** option is used, included deltas shall be listed following the notation, in the POSIX locale:

```
"Included:\n"
```

If the –**x** option is used, excluded deltas shall be listed following the notation, in the POSIX locale:

```
"Excluded:\n"
```

If the –**p** or –**L** options are specified, the standard output shall consist of the text retrieved from the SCCS file.

STDERR

The standard error shall be used only for diagnostic messages, except if the –**p** or –**L** options are specified, it shall include all informative messages normally sent to standard output.

OUTPUT FILES

Several auxiliary files may be created by *get*. These files are known generically as the **g-file**, **l-file**, **p-file**, and **z-file**. The letter before the hyphen is called the *tag*. An auxiliary filename shall be formed from the SCCS filename: the application shall ensure that the last component of all SCCS filenames is of the form **s.***module-name*; the auxiliary files shall be named by replacing the leading **s** with the tag. The **g-file** shall be an exception to this scheme: the **g-file** is named by removing the **s.** prefix. For example, for **s.xyz.c**, the auxiliary filenames would be **xyz.c**, **l.xyz.c**, **p.xyz.c**, and **z.xyz.c**, respectively.

The **g-file**, which contains the generated text, shall be created in the current directory (unless the –**p** option is used). A **g-file** shall be created in all cases, whether or not any lines of text were generated by the *get*. It shall be owned by the real user. If the –**k** option is used or implied, the **g-file** shall be writable by the owner only (read-only for everyone else); otherwise, it shall be read-only. Only the real user need have write permission in the current directory.

The **l-file** shall contain a table showing which deltas were applied in generating the retrieved text. The **l-file** shall be created in the current directory if the –**l** option is used; it shall be read-only and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the **l-file** shall have the following format:

```
"%c%c%cΔ%s\t%sΔ%s\n", <code1>, <code2>, <code3>,
    <SID>, <date-time>, <login>
```

where the entries are:

<code1>	A <space> if the delta was applied; `'*'` otherwise.
<code2>	A <space> if the delta was applied or was not applied and ignored; `'*'` if the delta was not applied and was not ignored.
<code3>	A character indicating a special reason why the delta was or was not applied: **I** Included. **X** Excluded. **C** Cut off (by a –**c** option).
<date-time>	Date and time (using the format of the *date* utility's `%y/%m/%d %T` conversion specification format) of creation.
<login>	Login name of person who created *delta*.

The comments and MR data shall follow on subsequent lines, indented one <tab>. A blank line shall terminate each entry.

The **p-file** shall be used to pass information resulting from a *get* with a −**e** option along to *delta*. Its contents shall also be used to prevent a subsequent execution of *get* with a −**e** option for the same SID until *delta* is executed or the joint edit flag, **j**, is set in the SCCS file. The **p-file** shall be created in the directory containing the SCCS file and the application shall ensure that the effective user has write permission in that directory. It shall be writable by owner only, and owned by the effective user. Each line in the **p-file** shall have the following format:

```
"%sΔ%sΔ%sΔ%s%s%s\n", <g-file SID>,
    <SID of new delta>, <login-name of real user>,
    <date-time>, <i-value>, <x-value>
```

where <*i-value*> uses the format " " if no −**i** option was specified, and shall use the format:

```
"Δ−i%s", <−i option option-argument>
```

if a −**i** option was specified and <*x-value*> uses the format " " if no −**x** option was specified, and shall use the format:

```
"Δ−x%s", <−x option option-argument>
```

if a −**x** option was specified. There can be an arbitrary number of lines in the **p-file** at any time; no two lines shall have the same new delta SID.

The **z-file** shall serve as a lock-out mechanism against simultaneous updates. Its contents shall be the binary process ID of the command (that is, *get*) that created it. The **z-file** shall be created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the **p-file** shall apply for the **z-file**. The **z-file** shall be created read-only.

EXTENDED DESCRIPTION

Determination of SCCS Identification String

SID* Specified	–b Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does not exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk successor	R.L	R.(L+1)
R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	R.L.B.S	R.L.(mB+1).1

* R, L, B, and S are the release, level, branch, and sequence components of the SID, respectively; m means maximum. Thus, for example, R.mL means ''the maximum level number within release R''; R.L.(mB+1).1 means ''the first sequence number on the new branch (that is, maximum branch number plus one) of level L within release R''. Note that if the SID specified is of the form R.L, R.L.B, or R.L.B.S, each of the specified components shall exist.

** hR is the highest existing release that is lower than the specified, nonexistent, release R.

*** This is used to force creation of the first delta in a new release.

† The –**b** option is effective only if the **b** flag is present in the file. An entry of '–' means ''irrelevant''.

‡ This case applies if the **d** (default SID) flag is not present in the file. If the **d** flag is present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

System Date and Time

When a **g-file** is generated, the creation time of deltas in the SCCS file may be taken into account. If any of these times are apparently in the future, the behavior is unspecified.

Identification Keywords

Identifying information shall be inserted into the text retrieved from the SCCS file by replacing identification keywords with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

%**M**%	Module name: either the value of the **m** flag in the file, or if absent, the name of the SCCS file with the leading **s.** removed.
%**I**%	SCCS identification (SID) (%**R**%.%**L**% or %**R**%.%**L**%.%**B**%.%**S**%) of the retrieved text.
%**R**%	Release.
%**L**%	Level.
%**B**%	Branch.
%**S**%	Sequence.
%**D**%	Current date (*YY*/*MM*/*DD*).
%**H**%	Current date (*MM*/*DD*/*YY*).
%**T**%	Current time (*HH*:*MM*:*SS*).
%**E**%	Date newest applied delta was created (*YY*/*MM*/*DD*).
%**G**%	Date newest applied delta was created (*MM*/*DD*/*YY*).
%**U**%	Time newest applied delta was created (*HH*:*MM*:*SS*).
%**Y**%	Module type: value of the **t** flag in the SCCS file.
%**F**%	SCCS filename.
%**P**%	SCCS absolute pathname.
%**Q**%	The value of the **q** flag in the file.
%**C**%	Current line number. This keyword is intended for identifying messages output by the program, such as ‘‘this should not have happened’’ type errors. It is not intended to be used on every line to provide sequence numbers.
%**Z**%	The four-character string `"@(#)"` recognizable by *what*.
%**W**%	A shorthand notation for constructing *what* strings: `%W%=%Z%%M%<tab>%I%`
%**A**%	Another shorthand notation for constructing *what* strings: `%A%=%Z%%Y%%M%%I%%Z%`

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Problems can arise if the system date and time have been modified (for example, put forward and then back again, or unsynchronized clocks across a network) and can also arise when different values of the *TZ* environment variable are used.

Problems of a similar nature can also arise for the operation of the *delta* utility, which compares the previous file body against the working file as part of its normal operation.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

The **–lp** option may be withdrawn in a future issue.

SEE ALSO

admin, *delta*, *prs*, *what*

CHANGE HISTORY

First released in Issue 2.

Issue 5

Correction to the first format string in STDOUT.

The interpretation of the *YY* component of the **–c** *cutoff* argument is noted.

Issue 6

The obsolescent SYNOPSIS is removed, removing the **–lp** option.

The Open Group Corrigendum U025/5 is applied, correcting text in the OPTIONS section.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

The Open Group Corrigendum U048/1 is applied.

The EXTENDED DESCRIPTION section is updated to make partial SID handling unspecified, reflecting common usage, and to clarify SID ranges as per The Open Group Base Resolution bwg2001-007.

The Open Group Interpretation PIN4C.00014 is applied.

New text is added to the EXTENDED DESCRIPTION and APPLICATION USAGE sections regarding how the system date and time may be taken into account, and the *TZ* environment variable is added to the ENVIRONMENT VARIABLES section as per The Open Group Base Resolution bwg2001-007.

NAME

getconf — get configuration values

SYNOPSIS

```
getconf [ −v specification ] system_var
```

```
getconf [ −v specification ] path_var pathname
```

DESCRIPTION

In the first synopsis form, the *getconf* utility shall write to the standard output the value of the variable specified by the *system_var* operand.

In the second synopsis form, the *getconf* utility shall write to the standard output the value of the variable specified by the *path_var* operand for the path specified by the *pathname* operand.

The value of each configuration variable shall be determined as if it were obtained by calling the function from which it is defined to be available by this volume of IEEE Std 1003.1-200x or by the System Interfaces volume of IEEE Std 1003.1-200x (see the OPERANDS section). The value shall reflect conditions in the current operating environment.

OPTIONS

The *getconf* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−v *specification*

Indicate a specific specification and version for which configuration variables shall be determined. If this option is not specified, the values returned correspond to an implementation default conforming compilation environment.

If the command:

```
getconf _POSIX_V6_ILP32_OFF32
```

does not write `"−1\n"` or `"undefined\n"` to standard output, then commands of the form:

```
getconf −v POSIX_V6_ILP32_OFF32 ...
```

determine values for configuration variables corresponding to the POSIX_V6_ILP32_OFF32 compilation environment specified in *c99* (on page 2413), EXTENDED DESCRIPTION.

If the command:

```
getconf _POSIX_V6_ILP32_OFFBIG
```

does not write `"−1\n"` or `"undefined\n"` to standard output, then commands of the form:

```
getconf −v POSIX_V6_ILP32_OFFBIG ...
```

determine values for configuration variables corresponding to the POSIX_V6_ILP32_OFFBIG compilation environment specified in *c99* (on page 2413), EXTENDED DESCRIPTION.

If the command:

```
getconf _POSIX_V6_LP64_OFF64
```

does not write `"−1\n"` or `"undefined\n"` to standard output, then commands of the form:

```
getconf −v POSIX_V6_LP64_OFF64 ...
```

determine values for configuration variables corresponding to the POSIX_V6_LP64_OFF64 compilation environment specified in *c99* (on page 2413), EXTENDED DESCRIPTION.

If the command:

```
getconf _POSIX_V6_LPBIG_OFFBIG
```

does not write `"−1\n"` or `"undefined\n"` to standard output, then commands of the form:

```
getconf −v POSIX_V6_LPBIG_OFFBIG ...
```

determine values for configuration variables corresponding to the POSIX_V6_LPBIG_OFFBIG compilation environment specified in *c99* (on page 2413), EXTENDED DESCRIPTION.

OPERANDS

The following operands shall be supported:

path_var A name of a configuration variable. All of the variables in the *pathconf*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x are supported and the implementation may add other local variables.

pathname A pathname for which the variable specified by *path_var* is to be determined.

system_var A name of a configuration variable. All of the variables in the *confstr*( ) and *sysconf*( ) functions defined in the System Interfaces volume of IEEE Std 1003.1-200x shall be supported and the implementation may add other local values.

When the symbol listed in the first column of the following table is used as the *system_var* operand, *getconf* yields the same value as *confstr*( ) when called with the value in the second column:

system_var	***confstr*( ) Name Value**
PATH	_CS_PATH
POSIX_V6_ILP32_OFF32_CFLAGS	_CS_POSIX_V6_ILP32_OFF32_CFLAGS
POSIX_V6_ILP32_OFF32_LDFLAGS	_CS_POSIX_V6_ILP32_OFF32_LDFLAGS
POSIX_V6_ILP32_OFF32_LIBS	_CS_POSIX_V6_ILP32_OFF32_LIBS
POSIX_V6_ILP32_OFF32_LINTFLAGS	_CS_POSIX_V6_ILP32_OFF32_LINTFLAGS
POSIX_V6_ILP32_OFFBIG_CFLAGS	_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
POSIX_V6_ILP32_OFFBIG_LDFLAGS	_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
POSIX_V6_ILP32_OFFBIG_LIBS	_CS_POSIX_V6_ILP32_OFFBIG_LIBS
POSIX_V6_ILP32_OFFBIG_LINTFLAGS	_CS_POSIX_V6_ILPBIG_OFF32_LINTFLAGS
POSIX_V6_LP64_OFF64_CFLAGS	_CS_POSIX_V6_LP64_OFF64_CFLAGS
POSIX_V6_LP64_OFF64_LDFLAGS	_CS_POSIX_V6_LP64_OFF64_LDFLAGS
POSIX_V6_LP64_OFF64_LIBS	_CS_POSIX_V6_LP64_OFF64_LIBS
POSIX_V6_LP64_OFF64_LINTFLAGS	_CS_POSIX_V6_LP64_OFF64_LINTFLAGS
POSIX_V6_LPBIG_OFFBIG_CFLAGS	_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
POSIX_V6_LPBIG_OFFBIG_LDFLAGS	_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
POSIX_V6_LPBIG_OFFBIG_LIBS	_CS_POSIX_V6_LPBIG_OFFBIG_LIBS

system_var	*confstr*( ) **Name Value**
POSIX_V6_LPBIG_OFFBIG_LINTFLAGS	_CS_POSIX_V6_LPBIG_OFFBIG_LINTFLAGS
POSIX_V6_WIDTH_RESTRICTED_ENVS	_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS
XSI XBS5_ILP32_OFF32_CFLAGS (**LEGACY**)	_CS_XBS5_ILP32_OFF32_CFLAGS
XBS5_ILP32_OFF32_LDFLAGS (**LEGACY**)	_CS_XBS5_ILP32_OFF32_LDFLAGS
XBS5_ILP32_OFF32_LIBS (**LEGACY**)	_CS_XBS5_ILP32_OFF32_LIBS
XBS5_ILP32_OFF32_LINTFLAGS (**LEGACY**)	_CS_XBS5_ILP32_OFF32_LINTFLAGS
XBS5_ILP32_OFFBIG_CFLAGS (**LEGACY**)	_CS_XBS5_ILP32_OFFBIG_CFLAGS
XBS5_ILP32_OFFBIG_LDFLAGS (**LEGACY**)	_CS_XBS5_ILP32_OFFBIG_LDFLAGS
XBS5_ILP32_OFFBIG_LIBS (**LEGACY**)	_CS_XBS5_ILP32_OFFBIG_LIBS
XBS5_ILP32_OFFBIG_LINTFLAGS (**LEGACY**)	_CS_XBS5_ILPBIG_OFF32_LINTFLAGS
XBS5_LP64_OFF64_CFLAGS (**LEGACY**)	_CS_XBS5_LP64_OFF64_CFLAGS
XBS5_LP64_OFF64_LDFLAGS (**LEGACY**)	_CS_XBS5_LP64_OFF64_LDFLAGS
XBS5_LP64_OFF64_LIBS (**LEGACY**)	_CS_XBS5_LP64_OFF64_LIBS
XBS5_LP64_OFF64_LINTFLAGS (**LEGACY**)	_CS_XBS5_LP64_OFF64_LINTFLAGS
XBS5_LPBIG_OFFBIG_CFLAGS (**LEGACY**)	_CS_XBS5_LPBIG_OFFBIG_CFLAGS
XBS5_LPBIG_OFFBIG_LDFLAGS (**LEGACY**)	_CS_XBS5_LPBIG_OFFBIG_LDFLAGS
XBS5_LPBIG_OFFBIG_LIBS (**LEGACY**)	_CS_XBS5_LPBIG_OFFBIG_LIBS
XBS5_LPBIG_OFFBIG_LINTFLAGS (**LEGACY**)	_CS_XBS5_LPBIG_OFFBIG_LINTFLAGS

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *getconf*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the specified variable is defined on the system and its value is described to be available from the *confstr*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, its value shall be written in the following format:

```
"%s\n", <value>
```

Otherwise, if the specified variable is defined on the system, its value shall be written in the following format:

```
"%d\n", <value>
```

If the specified variable is valid, but is undefined on the system, *getconf* shall write using the following format:

```
"undefined\n"
```

If the variable name is invalid or an error occurs, nothing shall be written to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The specified variable is valid and information about its current state was written successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

The following example illustrates the value of {NGROUPS_MAX}:

```
getconf NGROUPS_MAX
```

The following example illustrates the value of {NAME_MAX} for a specific directory:

```
getconf NAME_MAX /usr
```

The following example shows how to deal more carefully with results that might be unspecified:

```
if value=$(getconf PATH_MAX /usr); then
    if [ "$value" = "undefined" ]; then
        echo PATH_MAX in /usr is infinite.
    else
        echo PATH_MAX in /usr is $value.
    fi
else
    echo Error in getconf.
fi
```

Note that:

```
sysconf(_SC_POSIX_C_BIND);
```

and:

```
system("getconf POSIX2_C_BIND");
```

in a C program could give different answers. The *sysconf*( ) call supplies a value that corresponds to the conditions when the program was either compiled or executed, depending on the implementation; the *system*( ) call to *getconf* always supplies a value corresponding to conditions when the program is executed.

RATIONALE

The original need for this utility, and for the *confstr*( ) function, was to provide a way of finding the configuration-defined default value for the *PATH* environment variable. Since *PATH* can be modified by the user to include directories that could contain utilities replacing the standard utilities, shell scripts need a way to determine the system-supplied *PATH* environment variable value that contains the correct search path for the standard utilities. It was later suggested that access to the other variables described in this volume of IEEE Std 1003.1-200x could also be useful to applications.

This functionality of *getconf* would not be adequately subsumed by another command such as:

```
grep var /etc/conf
```

because such a strategy would provide correct values for neither those variables that can vary at runtime, nor those that can vary depending on the path.

Early proposal versions of *getconf* specified exit status 1 when the specified variable was valid, but not defined on the system. The output string `"undefined"` is now used to specify this case with exit code 0 because so many things depend on an exit code of zero when an invoked utility is successful.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, the System Interfaces volume of IEEE Std 1003.1-200x, *confstr*( ), *pathconf*( ), *sysconf*( )

CHANGE HISTORY

First released in Issue 4.

Issue 5

In the OPERANDS section:

- {NL_MAX} is changed to {NL_NMAX}.
- Entries beginning NL_ are deleted from the list of standard configuration variables.
- The list of variables previously marked UX is merged with the list marked EX.
- Operands are added to support new Option Groups.
- Operands are added so that *getconf* can determine supported programming environments.

Issue 6

The Open Group Corrigendum U029/4 is applied, correcting the example command in the last paragraph of the OPTIONS section.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- Operands are added to determine supported programming environments.

This reference page is updated for alignment with the ISO/IEC 9899: 1999 standard. Specifically, new macros for *c99* programming environments are introduced.

XSI marked *system_var* (XBS5_*) values are marked LEGACY.

NAME

getopts — parse utility options

SYNOPSIS

`getopts` *`optstring name`* `[`*`arg`*`...]`

DESCRIPTION

The *getopts* utility shall retrieve options and option-arguments from a list of parameters. It shall support the Utility Syntax Guidelines 3 to 10, inclusive, described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

Each time it is invoked, the *getopts* utility shall place the value of the next option in the shell variable specified by the *name* operand and the index of the next argument to be processed in the shell variable *OPTIND*. Whenever the shell is invoked, *OPTIND* shall be initialized to 1.

When the option requires an option-argument, the *getopts* utility shall place it in the shell variable *OPTARG*. If no option was found, or if the option that was found does not have an option-argument, *OPTARG* shall be unset.

If an option character not contained in the *optstring* operand is found where an option character is expected, the shell variable specified by *name* shall be set to the question-mark (`'?'`) character. In this case, if the first character in *optstring* is a colon (`':'`), the shell variable *OPTARG* shall be set to the option character found, but no output shall be written to standard error; otherwise, the shell variable *OPTARG* shall be unset and a diagnostic message shall be written to standard error. This condition shall be considered to be an error detected in the way arguments were presented to the invoking application, but shall be not an error in *getopts* processing.

If an option-argument is missing:

- If the first character of *optstring* is a colon, the shell variable specified by *name* shall be set to the colon character and the shell variable *OPTARG* shall be set to the option character found.
- Otherwise, the shell variable specified by *name* shall be set to the question-mark character, the shell variable *OPTARG* shall be unset, and a diagnostic message shall be written to standard error. This condition shall be considered to be an error detected in the way arguments were presented to the invoking application, but shall not be an error in *getopts* processing; a diagnostic message shall be written as stated, but the exit status shall be zero.

When the end of options is encountered, the *getopts* utility shall exit with a return value greater than zero; the shell variable *OPTIND* shall be set to the index of the first non-option-argument, where the first `"−−"` argument is considered to be an option-argument if there are no other non-option-arguments appearing before it, or the value `"$#"+1` if there are no non-option-arguments; the *name* variable shall be set to the question-mark character. Any of the following shall identify the end of options: the special option `"−−"`, finding an argument that does not begin with a `'−'`, or encountering an error.

The shell variables *OPTIND* and *OPTARG* shall be local to the caller of *getopts* and shall not be exported by default.

The shell variable specified by the *name* operand, *OPTIND* and *OPTARG* shall affect the current shell execution environment; see Section 2.12 (on page 2263).

If the application sets *OPTIND* to the value 1, a new set of parameters can be used: either the current positional parameters or new *arg* values. Any other attempt to invoke *getopts* multiple times in a single shell execution environment with parameters (positional parameters or *arg* operands) that are not the same in all invocations, or with an *OPTIND* value modified to be a value other than 1, produces unspecified results.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

optstring	A string containing the option characters recognized by the utility invoking *getopts*. If a character is followed by a colon, the option shall be expected to have an argument, which should be supplied as a separate argument. Applications should specify an option character and its option-argument as separate arguments, but *getopts* shall interpret the characters following an option character requiring arguments as an argument whether or not this is done. An explicit null option-argument need not be recognized if it is not supplied as a separate argument when *getopts* is invoked. (See also the *getopt*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x.) The characters question-mark and colon shall not be used as option characters by an application. The use of other option characters that are not alphanumeric produces unspecified results. If the option-argument is not supplied as a separate argument from the option character, the value in *OPTARG* shall be stripped of the option character and the '−'. The first character in *optstring* determines how *getopts* behaves if an option character is not known or an option-argument is missing.
name	The name of a shell variable that shall be set by the *getopts* utility to the option character that was found.

The *getopts* utility by default shall parse positional parameters passed to the invoking shell procedure. If *arg*s are given, they shall be parsed instead of the positional parameters.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *getopts*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
OPTIND	This variable shall be used by the *getopts* utility as the index of the next argument to be processed.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Whenever an error is detected and the first character in the *optstring* operand is not a colon (`':'`), a diagnostic message shall be written to standard error with the following information in an unspecified format:

- The invoking program name shall be identified in the message. The invoking program name shall be the value of the shell special parameter 0 (see Section 2.5.2 (on page 2235)) at the time the *getopts* utility is invoked. A name equivalent to:

  ```
  basename "$0"
  ```

  may be used.

- If an option is found that was not specified in *optstring*, this error is identified and the invalid option character shall be identified in the message.

- If an option requiring an option-argument is found, but an option-argument is not found, this error shall be identified and the invalid option character shall be identified in the message.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 An option, specified or unspecified by *optstring*, was found.

>0 The end of options was encountered or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *getopts* affects the current shell execution environment, it is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(getopts abc value "$@")
nohup getopts ...
find . −exec getopts ... \;
```

it does not affect the shell variables in the caller's environment.

Note that shell functions share *OPTIND* with the calling shell even though the positional parameters are changed. If the calling shell and any of its functions uses *getopts* to parse arguments, the results are unspecified.

EXAMPLES

The following example script parses and displays its arguments:

```
aflag=
bflag=
```

```
while getopts ab: name
do
    case $name in
    a)    aflag=1;;
    b)    bflag=1
          bval="$OPTARG";;
    ?)   printf "Usage: %s: [-a] [-b value] args\n" $0
          exit 2;;
    esac
done
if [ ! -z "$aflag" ]; then
    printf "Option -a specified\n"
fi
if [ ! -z "$bflag" ]; then
    printf 'Option -b "%s" specified\n' "$bval"
fi
shift $(($OPTIND - 1))
printf "Remaining arguments are: %s\n" "$*"
```

RATIONALE

The *getopts* utility was chosen in preference to the System V *getopt* utility because *getopts* handles option-arguments containing <blank>s.

The *OPTARG* variable is not mentioned in the ENVIRONMENT VARIABLES section because it does not affect the execution of *getopts*; it is one of the few ''output-only'' variables used by the standard utilities.

The colon is not allowed as an option character because that is not historical behavior, and it violates the Utility Syntax Guidelines. The colon is now specified to behave as in the KornShell version of the *getopts* utility; when used as the first character in the *optstring* operand, it disables diagnostics concerning missing option-arguments and unexpected option characters. This replaces the use of the *OPTERR* variable that was specified in an early proposal.

The formats of the diagnostic messages produced by the *getopts* utility and the *getopt*() function are not fully specified because implementations with superior (''friendlier'') formats objected to the formats used by some historical implementations. The standard developers considered it important that the information in the messages used be uniform between *getopts* and *getopt*(). Exact duplication of the messages might not be possible, particularly if a utility is built on another system that has a different *getopt*() function, but the messages must have specific information included so that the program name, invalid option character, and type of error can be distinguished by a user.

Only a rare application program intercepts a *getopts* standard error message and wants to parse it. Therefore, implementations are free to choose the most usable messages they can devise. The following formats are used by many historical implementations:

```
"%s: illegal option -- %c\n", <program name>, <option character>
```

```
"%s: option requires an argument -- %c\n", <program name>, \
    <option character>
```

Historical shells with built-in versions of *getopt*() or *getopts* have used different formats, frequently not even indicating the option character found in error.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *getopt*( )

CHANGE HISTORY

First released in Issue 4.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

grep — search a file for a pattern

SYNOPSIS

```
grep [−E| −F][−c| −l| −q][−insvx] −e pattern_list...
    [−f pattern_file]...[file...]

grep [−E| −F][−c| −l| −q][−insvx][−e pattern_list]...
    −f pattern_file...[file...]

grep [−E| −F][−c| −l| −q][−insvx] pattern_list[file...]
```

DESCRIPTION

The *grep* utility shall search the input files, selecting lines matching one or more patterns; the types of patterns are controlled by the options specified. The patterns are specified by the **−e** option, **−f** option, or the *pattern_list* operand. The *pattern_list*'s value shall consist of one or more patterns separated by <newline>s; the *pattern_file*'s contents shall consist of one or more patterns terminated by <newline>. By default, an input line shall be selected if any pattern, treated as an entire basic regular expression (BRE) as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions, matches any part of the line excluding the terminating <newline>; a null BRE shall match every line. By default, each selected input line shall be written to the standard output.

Regular expression matching shall be based on text lines. Since a <newline> separates or terminates patterns (see the **−e** and **−f** options below), regular expressions cannot contain a <newline>. Similarly, since patterns are matched against individual lines (excluding the terminating <newline>s) of the input, there is no way for a pattern to match a <newline> found in the input.

OPTIONS

The *grep* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−E Match using extended regular expressions. Treat each pattern specified as an ERE, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.4, Extended Regular Expressions. If any entire ERE pattern matches some part of an input line excluding the terminating <newline>, the line shall be matched. A null ERE shall match every line.

−F Match using fixed strings. Treat each pattern specified as a string instead of a regular expression. If an input line contains any of the patterns as a contiguous sequence of bytes, the line shall be matched. A null string shall match every line.

−c Write only a count of selected lines to standard output.

−e *pattern_list*

Specify one or more patterns to be used during the search for input. The application shall ensure that patterns in *pattern_list* are separated by a <newline>. A null pattern can be specified by two adjacent <newline>s in *pattern_list*. Unless the **−E** or **−F** option is also specified, each pattern shall be treated as a BRE, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions. Multiple **−e** and **−f** options shall be accepted by the *grep* utility. All of the specified patterns shall be used when matching lines, but the order of evaluation is unspecified.

–f *pattern_file*	Read one or more patterns from the file named by the pathname *pattern_file*. Patterns in *pattern_file* shall be terminated by a <newline>. A null pattern can be specified by an empty line in *pattern_file*. Unless the **–E** or **–F** option is also specified, each pattern shall be treated as a BRE, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions.
–i	Perform pattern matching in searches without regard to case; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.2, Regular Expression General Requirements.
–l	(The letter ell.) Write only the names of files containing selected lines to standard output. Pathnames shall be written once per file searched. If the standard input is searched, a pathname of `"(standard input)"` shall be written, in the POSIX locale. In other locales, `"standard input"` may be replaced by something more appropriate in those locales.
–n	Precede each output line by its relative line number in the file, each file starting at line 1. The line number counter shall be reset for each file processed.
–q	Quiet. Nothing shall be written to the standard output, regardless of matching lines. Exit with zero status if an input line is selected.
–s	Suppress the error messages ordinarily written for nonexistent or unreadable files. Other error messages shall not be suppressed.
–v	Select lines not matching any of the specified patterns. If the **–v** option is not specified, selected lines shall be those that match any of the specified patterns.
–x	Consider only input lines that use all characters in the line excluding the terminating <newline> to match an entire fixed string or regular expression to be matching lines.

OPERANDS

The following operands shall be supported:

pattern_list	Specify one or more patterns to be used during the search for input. This operand shall be treated as if it were specified as **–e** *pattern_list*.
file	A pathname of a file to be searched for the patterns. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *grep*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes within regular expressions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the −**l** option is in effect, and the −**q** option is not, the following shall be written for each file containing at least one selected input line:

```
"%s\n", <file>
```

Otherwise, if more than one *file* argument appears, and −**q** is not specified, the *grep* utility shall prefix each output line by:

```
"%s:", <file>
```

The remainder of each output line shall depend on the other options specified:

- If the −**c** option is in effect, the remainder of each output line shall contain:

  ```
  "%d\n", <count>
  ```

- Otherwise, if −**c** is not in effect and the −**n** option is in effect, the following shall be written to standard output:

  ```
  "%d:", <line number>
  ```

- Finally, the following shall be written to standard output:

  ```
  "%s", <selected-line contents>
  ```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	One or more lines were selected.
1	No lines were selected.
>1	An error occurred.

CONSEQUENCES OF ERRORS

If the **−q** option is specified, the exit status shall be zero if an input line is selected, even if an error was detected. Otherwise, default actions shall be performed.

APPLICATION USAGE

Care should be taken when using characters in *pattern_list* that may also be meaningful to the command interpreter. It is safest to enclose the entire *pattern_list* argument in single quotes:

```
'...'
```

The **−e** *pattern_list* option has the same effect as the *pattern_list* operand, but is useful when *pattern_list* begins with the hyphen delimiter. It is also useful when it is more convenient to provide multiple patterns as separate arguments.

Multiple **−e** and **−f** options are accepted and *grep* uses all of the patterns it is given while matching input text lines. (Note that the order of evaluation is not specified. If an implementation finds a null string as a pattern, it is allowed to use that pattern first, matching every line, and effectively ignore any other patterns.)

The **−q** option provides a means of easily determining whether or not a pattern (or string) exists in a group of files. When searching several files, it provides a performance improvement (because it can quit as soon as it finds the first match) and requires less care by the user in choosing the set of files to supply as arguments (because it exits zero if it finds a match even if *grep* detected an access or read error on earlier *file* operands).

EXAMPLES

1. To find all uses of the word `"Posix"` (in any case) in file **text.mm** and write with line numbers:

   ```
   grep −i −n posix text.mm
   ```

2. To find all empty lines in the standard input:

   ```
   grep ^$
   ```

   or:

   ```
   grep −v .
   ```

3. Both of the following commands print all lines containing strings `"abc"` or `"def"` or both:

   ```
   grep −E 'abc|def'
   ```

   ```
   grep −F 'abc|def'
   ```

4. Both of the following commands print all lines matching exactly `"abc"` or `"def"`:

   ```
   grep −E '^abc$|^def$'
   ```

   ```
   grep −F −x 'abc|def'
   ```

RATIONALE

This *grep* has been enhanced in an upward-compatible way to provide the exact functionality of the historical *egrep* and *fgrep* commands as well. It was the clear intention of the standard developers to consolidate the three *grep*s into a single command.

The old *egrep* and *fgrep* commands are likely to be supported for many years to come as implementation extensions, allowing historical applications to operate unmodified.

Historical implementations usually silently ignored all but one of multiply-specified **−e** and **−f** options, but were not consistent as to which specification was actually used.

The **–b** option was omitted from the OPTIONS section because block numbers are implementation-defined.

The System V restriction on using – to mean standard input was omitted.

A definition of action taken when given a null BRE or ERE is specified. This is an error condition in some historical implementations.

The **–l** option previously indicated that its use was undefined when no files were explicitly named. This behavior was historical and placed an unnecessary restriction on future implementations. It has been removed.

The historical BSD *grep* **–s** option practice is easily duplicated by redirecting standard output to **/dev/null**. The **–s** option required here is from System V.

The **–x** option, historically available only with *fgrep*, is available here for all of the non-obsolescent versions.

FUTURE DIRECTIONS

None.

SEE ALSO

sed

CHANGE HISTORY

First released in Issue 2.

Issue 6

The Open Group Corrigendum U029/5 is applied, correcting the SYNOPSIS.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

hash — remember or report utility locations

SYNOPSIS

XSI

```
hash [utility...]

hash −r
```

DESCRIPTION

The *hash* utility shall affect the way the current shell environment remembers the locations of utilities found as described in Section 2.9.1.1 (on page 2249). Depending on the arguments specified, it shall add utility locations to its list of remembered locations or it shall purge the contents of the list. When no arguments are specified, it shall report on the contents of the list.

Utilities provided as built-ins to the shell shall not be reported by *hash*.

OPTIONS

The *hash* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−r	Forget all previously remembered utility locations.

OPERANDS

The following operand shall be supported:

utility	The name of a utility to be searched for and added to the list of remembered locations. If *utility* contains one or more slashes, the results are unspecified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *hash*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the location of *utility*, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output of *hash* shall be used when no arguments are specified. Its format is unspecified, but includes the pathname of each utility in the list of remembered locations for the current shell environment. This list shall consist of those utilities named in previous *hash* invocations that have been invoked, and may contain those invoked and found through the normal command search process.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *hash* affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in a separate utility execution environment, such as one of the following:

```
nohup hash −r
find . −type f | xargs hash
```

it does not affect the command search process of the caller's environment.

The *hash* utility may be implemented as an alias—for example, *alias* −**t** −, in which case utilities found through normal command search are not listed by the *hash* command.

The effects of *hash* −**r** can also be achieved portably by resetting the value of *PATH*; in the simplest form, this can be:

```
PATH="$PATH"
```

The use of *hash* with *utility* names is unnecessary for most applications, but may provide a performance improvement on a few implementations; normally, the hashing process is included by default.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.9.1.1 (on page 2249)

CHANGE HISTORY

First released in Issue 2.

NAME

head — copy the first part of files

SYNOPSIS

```
head [−n number][file...]
```

DESCRIPTION

The *head* utility shall copy its input files to the standard output, ending the output for each file at a designated point.

Copying shall end at the point in each input file indicated by the **−n** *number* option. The option-argument *number* shall be counted in units of lines.

OPTIONS

The *head* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−n *number*	The first *number* lines of each input file shall be copied to standard output. The application shall ensure that the *number* option-argument is a positive decimal integer.

If no options are specified, *head* shall act as if **−n 10** had been specified.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

Input files shall be text files, but the line length is not restricted to {LINE_MAX} bytes.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *head*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall contain designated portions of the input files.

If multiple *file* operands are specified, *head* shall precede the output for each with the header:

```
"\n==> %s <==\n", <pathname>
```

except that the first header written shall not include the initial <newline>.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The obsolescent −*number* form is withdrawn in this version. Applications should use the −**n** *number* option.

EXAMPLES

To write the first ten lines of all files (except those with a leading period) in the directory:

```
head *
```

RATIONALE

Although it is possible to simulate *head* with *sed* 10q for a single file, the standard developers decided that the popularity of *head* on historical BSD systems warranted its inclusion alongside *tail*.

This standard version of *head* follows the Utility Syntax Guidelines. The −**n** option was added to this new interface so that *head* and *tail* would be more logically related.

There is no −**c** option (as there is in *tail*) because it is not historical practice and because other utilities in this volume of IEEE Std 1003.1-200x provide similar functionality.

FUTURE DIRECTIONS

None.

SEE ALSO

sed, *tail*

CHANGE HISTORY

First released in Issue 4.

Issue 6

The obsolescent −**number** form is withdrawn.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

iconv — codeset conversion

SYNOPSIS

```
iconv [−cs] −f fromcode −t tocode [file ...]

iconv −l
```

DESCRIPTION

The *iconv* utility shall convert the encoding of characters in *file* from one codeset to another and write the results to standard output.

When the options indicate that charmap files are used to specify the codesets (see OPTIONS), the codeset conversion shall be accomplished by performing a logical join on the symbolic character names in the two charmaps. The implementation need not support the use of charmap files for codeset conversion unless the POSIX2_LOCALEDEF symbol is defined on the system.

OPTIONS

The *iconv* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c Omit any invalid characters from the output. When **−c** is not used, the results of encountering invalid characters in the input stream (either those that are not valid members of the *fromcode* or those that have no corresponding value in *tocode*) shall be specified in the system documentation. The presence or absence of **−c** shall not affect the exit status of *iconv*.

−f *fromcode* Identify the codeset of the input file. If the option-argument contains a slash character, *iconv* shall attempt to use it as the pathname of a charmap file, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.4, Character Set Description File. If the pathname does not represent a valid, readable charmap file, the results are undefined. If the option-argument does not contain a slash, it shall be considered the name of one of the codeset descriptions provided by the system, in an unspecified format. The valid values of the option-argument without a slash are implementation-defined. If this option is omitted, the codeset of the current locale shall be used.

−l Write all supported *fromcode* and *tocode* values to standard output in an unspecified format.

−s Suppress any messages written to standard error concerning invalid characters. When **−s** is not used, the results of encountering invalid characters in the input stream (either those that are not valid members of the *fromcode* or those that have no corresponding value in *tocode*) shall be specified in the system documentation. The presence or absence of **−s** shall not affect the exit status of *iconv*.

−t *tocode* Identify the codeset to be used for the output file. The semantics shall be equivalent to the **−f** *fromcode* option. |

If either **−f** or **−t** represents a charmap file, but the other does not (or is omitted), or both **−f** and **−t** are omitted, the results are undefined.

OPERANDS

The following operand shall be supported:

file A pathname of an input file. If no *file* operands are specified, or if a *file* operand is `'−'`, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is '−'.

INPUT FILES

The input file shall be a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *iconv*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments). During translation of the file, this variable is superseded by the use of the *fromcode* option-argument.
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **−l** option is used, the standard output shall contain all supported *fromcode* and *tocode* values, written in an unspecified format.

When the **−l** option is not used, the standard output shall contain the sequence of characters read from the input files, translated to the specified codeset. Nothing else shall be written to the standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The user must ensure that both charmap files use the same symbolic names for characters the two codesets have in common.

EXAMPLES

The following example converts the contents of file **mail.x400** from the ISO/IEC 6937:1994 standard codeset to the ISO/IEC 8859-1:1998 standard codeset, and stores the results in file **mail.local**:

```
iconv −f IS6937 −t IS8859 mail.x400 > mail.local
```

RATIONALE

The *iconv* utility can be used portably only when the user provides two charmap files as option-arguments. This is because a single charmap provided by the user cannot reliably be joined with the names in a system-provided character set description. The valid values for *fromcode* and *tocode* are implementation-defined and do not have to have any relation to the charmap mechanisms. As an aid to interactive users, the **−l** option was adopted from the Plan 9 operating system. It writes information concerning these implementation-defined values. The format is unspecified because there are many possible useful formats that could be chosen, such as a matrix of valid combinations of *fromcode* and *tocode*. The **−l** option is not intended for shell script usage; conforming applications will have to use charmaps.

FUTURE DIRECTIONS

None.

SEE ALSO

gencat

CHANGE HISTORY

First released in Issue 3.

Issue 6

This utility has been rewritten to align with the IEEE P1003.2b draft standard. Specifically, the ability to use charmap files for conversion has been added.

NAME

id — return user identity

SYNOPSIS

```
id [user]
id -G[-n] [user]
id -g[-nr] [user]
id -u[-nr] [user]
```

DESCRIPTION

If no *user* operand is provided, the *id* utility shall write the user and group IDs and the corresponding user and group names of the invoking process to standard output. If the effective and real IDs do not match, both shall be written. If multiple groups are supported by the underlying system (see the description of {NGROUPS_MAX} in the System Interfaces volume of IEEE Std 1003.1-200x), the supplementary group affiliations of the invoking process shall also be written.

If a *user* operand is provided and the process has the appropriate privileges, the user and group IDs of the selected user shall be written. In this case, effective IDs shall be assumed to be identical to real IDs. If the selected user has more than one allowable group membership listed in the group database, these shall be written in the same manner as the supplementary groups described in the preceding paragraph.

OPTIONS

The *id* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−G	Output all different group IDs (effective, real, and supplementary) only, using the format `"%u\n"`. If there is more than one distinct group affiliation, output each such affiliation, using the format `" %u"`, before the <newline> is output.
−g	Output only the effective group ID, using the format `"%u\n"`.
−n	Output the name in the format `"%s"` instead of the numeric ID using the format `"%u"`.
−r	Output the real ID instead of the effective ID.
−u	Output only the effective user ID, using the format `"%u\n"`.

OPERANDS

The following operand shall be supported:

user	The login name for which information is to be written.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *id*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables

used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The following formats shall be used when the *LC_MESSAGES* locale category specifies the POSIX locale. In other locales, the strings *uid*, *gid*, *euid*, *egid*, and *groups* may be replaced with more appropriate strings corresponding to the locale.

```
"uid=%u(%s) gid=%u(%s)\n", <real user ID>, <user-name>,
    <real group ID>, <group-name>
```

If the effective and real user IDs do not match, the following shall be inserted immediately before the `'\n'` character in the previous format:

```
" euid=%u(%s)"
```

with the following arguments added at the end of the argument list:

```
<effective user ID>, <effective user-name>
```

If the effective and real group IDs do not match, the following shall be inserted directly before the `'\n'` character in the format string (and after any addition resulting from the effective and real user IDs not matching):

```
" egid=%u(%s)"
```

with the following arguments added at the end of the argument list:

```
<effective group-ID>, <effective group name>
```

If the process has supplementary group affiliations or the selected user is allowed to belong to multiple groups, the first shall be added directly before the <newline> in the format string:

```
" groups=%u(%s)"
```

with the following arguments added at the end of the argument list:

```
<supplementary group ID>, <supplementary group name>
```

and the necessary number of the following added after that for any remaining supplementary group IDs:

```
",%u(%s)"
```

and the necessary number of the following arguments added at the end of the argument list:

```
<supplementary group ID>, <supplementary group name>
```

If any of the user ID, group ID, effective user ID, effective group ID, or supplementary/multiple group IDs cannot be mapped by the system into printable user or group names, the corresponding ("%s") and *name* argument shall be omitted from the corresponding format string.

When any of the options are specified, the output format shall be as described in the OPTIONS section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Output produced by the −**G** option and by the default case could potentially produce very long lines on systems that support large numbers of supplementary groups. (On systems with user and group IDs that are 32-bit integers and with group names with a maximum of 8 bytes per name, 93 supplementary groups plus distinct effective and real group and user IDs could theoretically overflow the 2 048-byte {LINE_MAX} text file line limit on the default output case. It would take about 186 supplementary groups to overflow the 2 048-byte barrier using *id* −**G**). This is not expected to be a problem in practice, but in cases where it is a concern, applications should consider using *fold* −**s** before postprocessing the output of *id*.

EXAMPLES

None.

RATIONALE

The functionality provided by the 4 BSD *groups* utility can be simulated using:

```
id −Gn [ user ]
```

The 4 BSD command *groups* was considered, but it was not included because it did not provide the functionality of the *id* utility of the SVID. Also, it was thought that it would be easier to modify *id* to provide the additional functionality necessary to systems with multiple groups than to invent another command.

The options −**u**, −**g**, −**n**, and −**r** were added to ease the use of *id* with shell commands substitution. Without these options it is necessary to use some preprocessor such as *sed* to select the desired piece of information. Since output such as that produced by:

```
id −u −n
```

is frequently wanted, it seemed desirable to add the options.

FUTURE DIRECTIONS

None.

SEE ALSO

fold, *logname*, *who*, the System Interfaces volume of IEEE Std 1003.1-200x, *getgid*( ), *getgroups*( ), *getuid*( )

CHANGE HISTORY

First released in Issue 2.

NAME

ipcrm — remove an XSI message queue, semaphore set, or shared memory segment identifier

SYNOPSIS

XSI
```
ipcrm [ −q msgid | −Q msgkey | −s semid | −S semkey |
    −m shmid | −M shmkey ] ...
```

DESCRIPTION

The *ipcrm* utility shall remove zero or more message queues, semaphore sets, or shared memory segments. The interprocess communication facilities to be removed are specified by the options.

Only a user with appropriate privilege shall be allowed to remove an interprocess communication facility that was not created by or owned by the user invoking *ipcrm*.

OPTIONS

The *ipcrm* facility supports the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−q *msgid* Remove the message queue identifier *msgid* from the system and destroy the message queue and data structure associated with it.

−m *shmid* Remove the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it shall be destroyed after the last detach.

−s *semid* Remove the semaphore identifier *semid* from the system and destroy the set of semaphores and data structure associated with it.

−Q *msgkey* Remove the message queue identifier, created with key *msgkey*, from the system and destroy the message queue and data structure associated with it.

−M *shmkey* Remove the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it shall be destroyed after the last detach.

−S *semkey* Remove the semaphore identifier, created with key *semkey*, from the system and destroy the set of semaphores and data structure associated with it.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ipcrm*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ipcs, the System Interfaces volume of IEEE Std 1003.1-200x, *msgctl*( ), *semctl*( ), *shmctl*( )

CHANGE HISTORY

First released in Issue 5.

NAME

ipcs — report XSI interprocess communication facilities status

SYNOPSIS

XSI

```
ipcs [−qms][−a | −bcopt]
```

DESCRIPTION

The *ipcs* utility shall write information about active interprocess communication facilities.

Without options, information shall be written in short format for message queues, shared memory segments, and semaphores sets that are currently active in the system. Otherwise, the information that is displayed is controlled by the options specified.

OPTIONS

The *ipcs* facility supports the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The *ipcs* utility accepts the following options:

−q	Write information about active message queues.
−m	Write information about active shared memory segments.
−s	Write information about active semaphores sets.

If **−q**, **−m**, or **−s** are specified, only information about those facilities shall be written. If none of these three are specified, information about all three shall be written subject to the following options:

−a	Use all print options. (This is a shorthand notation for **−b**, **−c**, **−o**, **−p**, and **−t**.)
−b	Write information on maximum allowable size. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.)
−c	Write creator's user name and group name; see below.
−o	Write information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues, and number of processes attached to shared memory segments.)
−p	Write process number information. (Process ID of last process to send a message and process ID of last process to receive a message on message queues, process ID of creating process, and process ID of last process to attach or detach on shared memory segments.)
−t	Write time information. (Time of the last control operation that changed the access permissions for all facilities, time of last *msgsnd*( ) and *msgrcv*( ) operations on message queues, time of last *shmat*( ) and *shmdt*( ) operations on shared memory, and time of last *semop*( ) operation on semaphores.)

OPERANDS

None.

STDIN

Not used.

INPUT FILES

- The group database
- The user database

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ipcs*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
TZ	Determine the timezone for the date and time strings written by *ipcs*. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

An introductory line shall be written with the format:

```
"IPC status from %s as of %s\n", <source>, <date>
```

where <*source*> indicates the source used to gather the statistics and <*date*> is the information that would be produced by the *date* command when invoked in the POSIX locale.

The *ipcs* utility then shall create up to three reports depending upon the −**q**, −**m**, and −**s** options. The first report shall indicate the status of message queues, the second report shall indicate the status of shared memory segments, and the third report shall indicate the status of semaphore sets.

If the corresponding facility is not installed or has not been used since the last reboot, then the report shall be written out in the format:

```
"%s facility not in system.\n", <facility>
```

where <*facility*> is *Message Queue*, *Shared Memory*, or *Semaphore*, as appropriate. If the facility has been installed and has been used since the last reboot, column headings separated by one or more spaces and followed by a <newline> shall be written as indicated below followed by the facility name written out using the format:

```
"%s:\n", <facility>
```

where <*facility*> is *Message Queues*, *Shared Memory*, or *Semaphores*, as appropriate. On the second and third reports the column headings need not be written if the last column headings written already provide column headings for all information in that report.

The column headings provided in the first column below and the meaning of the information in those columns shall be given in order below; the letters in parentheses indicate the options that shall cause the corresponding column to appear; ''all'' means that the column shall always appear. Each column is separated by one or more <space>s. Note that these options only determine what information is provided for each report; they do not determine which reports are written.

T (all) Type of facility:

`q`	Message queue.
`m`	Shared memory segment.
`s`	Semaphore.

This field is a single character written using the format `%c`.

ID (all) The identifier for the facility entry. This field shall be written using the format `%d`.

KEY (all) The key used as an argument to *msgget*( ), *semget*( ), or *shmget*( ) to create the facility entry.

Note: The key of a shared memory segment is changed to IPC_PRIVATE when the segment has been removed until all processes attached to the segment detach it.

This field shall be written using the format `0x%x`.

MODE (all) The facility access modes and flags. The mode shall consist of 11 characters that are interpreted as follows.

The first character shall be:

`S`	If a process is waiting on a *msgsnd*( ) operation.
–	If the above is not true.

The second character shall be:

`R`	If a process is waiting on a *msgrcv*( ) operation.
`C` or –	If the associated shared memory segment is to be cleared when the first attach operation is executed.
–	If none of the above is true.

The next nine characters shall be interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the usergroup of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates permission to write or alter the facility entry, and the last character is a minus sign (`'−'`).

The permissions shall be indicated as follows:

r	If read permission is granted.
w	If write permission is granted.
a	If alter permission is granted.
–	If the indicated permission is not granted.

The first character following the permissions specifies if there is an alternate or additional access control method associated with the facility. If there is no alternate or additional access control method associated with the facility, a single <space> shall be written; otherwise, another printable character is written.

OWNER (all) The user name of the owner of the facility entry. If the user name of the owner is found in the user database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the user ID of the owner shall be written using the format `%d`.

GROUP (all) The group name of the owner of the facility entry. If the group name of the owner is found in the group database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the group ID of the owner shall be written using the format `%d`.

The following nine columns shall be only written out for message queues:

CREATOR (**a**,**c**) The user name of the creator of the facility entry. If the user name of the creator is found in the user database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the user ID of the creator shall be written using the format `%d`.

CGROUP (**a**,**c**) The group name of the creator of the facility entry. If the group name of the creator is found in the group database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the group ID of the creator shall be written using the format `%d`.

CBYTES (**a**,**o**) The number of bytes in messages currently outstanding on the associated message queue. This field shall be written using the format `%d`.

QNUM (**a**,**o**) The number of messages currently outstanding on the associated message queue. This field shall be written using the format `%d`.

QBYTES (**a**,**b**) The maximum number of bytes allowed in messages outstanding on the associated message queue. This field shall be written using the format `%d`.

LSPID (**a**,**p**) The process ID of the last process to send a message to the associated queue. This field shall be written using the format:

`"%d", <pid>`

where <*pid*> is 0 if no message has been sent to the corresponding message queue; otherwise, <*pid*> shall be the process ID of the last process to send a message to the queue.

LRPID (**a**,**p**) The process ID of the last process to receive a message from the associated queue. This field shall be written using the format:

`"%d", <pid>`

where <*pid*> is 0 if no message has been received from the corresponding message queue; otherwise, <*pid*> shall be the process ID of the last process to receive a message from the queue.

STIME (**a**,**t**) The time the last message was sent to the associated queue. If a message has been sent to the corresponding message queue, the hour, minute, and second of the last time a message was sent to the queue shall be written using the format `%d:%2.2d:%2.2d`. Otherwise, the format `" no-entry"` shall be written.

RTIME (**a**,**t**) The time the last message was received from the associated queue. If a message has been received from the corresponding message queue, the hour, minute, and second of the last time a message was received from the queue shall be written using the format `%d:%2.2d:%2.2d`. Otherwise, the format `" no-entry"` shall be written.

The following eight columns shall be only written out for shared memory segments.

CREATOR (**a**,**c**) The user of the creator of the facility entry. If the user name of the creator is found in the user database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the user ID of the creator shall be written using the format `%d`.

CGROUP (**a**,**c**) The group name of the creator of the facility entry. If the group name of the creator is found in the group database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the group ID of the creator shall be written using the format `%d`.

NATTCH (**a**,**o**) The number of processes attached to the associated shared memory segment. This field shall be written using the format `%d`.

SEGSZ (**a**,**b**) The size of the associated shared memory segment. This field shall be written using the format `%d`.

CPID (**a**,**p**) The process ID of the creator of the shared memory entry. This field shall be written using the format `%d`.

LPID (**a**,**p**) The process ID of the last process to attach or detach the shared memory segment. This field shall be written using the format:

`"%d", <pid>`

where <*pid*> is 0 if no process has attached the corresponding shared memory segment; otherwise, <*pid*> shall be the process ID of the last process to attach or detach the segment.

ATIME (**a**,**t**) The time the last attach on the associated shared memory segment was completed. If the corresponding shared memory segment has ever been attached, the hour, minute, and second of the last time the segment was attached shall be written using the format `%d:%2.2d:%2.2d`. Otherwise, the format `" no-entry"` shall be written.

DTIME (**a**,**t**) The time the last detach on the associated shared memory segment was completed. If the corresponding shared memory segment has ever been detached, the hour, minute, and second of the last time the segment was detached shall be written using the format `%d:%2.2d:%2.2d`. Otherwise, the format `" no-entry"` shall be written.

The following four columns shall be only written out for semaphore sets:

CREATOR (**a**,**c**) The user of the creator of the facility entry. If the user name of the creator is found in the user database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the user ID of the creator shall be written using the format `%d`.

CGROUP (**a**,**c**) The group name of the creator of the facility entry. If the group name of the creator is found in the group database, at least the first eight column positions of the name shall be written using the format `%s`. Otherwise, the group ID of the creator shall be written using the format `%d`.

NSEMS	(**a**,**b**)	The number of semaphores in the set associated with the semaphore entry. This field shall be written using the format `%d`.
OTIME	(**a**,**t**)	The time the last semaphore operation on the set associated with the semaphore entry was completed. If a semaphore operation has ever been performed on the corresponding semaphore set, the hour, minute, and second of the last semaphore operation on the semaphore set shall be written using the format `%d:%2.2d:%2.2d`. Otherwise, the format `" no-entry"` shall be written.

The following column shall be written for all three reports when it is requested:

CTIME	(**a**,**t**)	The time the associated entry was created or changed. The hour, minute, and second of the time when the associated entry was created shall be written using the format `%d:%2.2d:%2.2d`.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Things can change while *ipcs* is running; the information it gives is guaranteed to be accurate only when it was retrieved.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *msgop*( ), *msgrcv*( ), *msgsnd*( ), *semget*( ), *semop*( ), *shmat*( ), *shmdt*( ), *shmget*( ), *shmop*( )

CHANGE HISTORY

First released in Issue 5.

Issue 6

The Open Group Corrigendum U020/1 is applied, correcting the SYNOPSIS.

The Open Group Corrigendum U032/1 and U032/2 are applied, clarifying the output format.

The Open Group Base Resolution bwg98-004 is applied.

NAME

jobs — display status of jobs in the current session

SYNOPSIS

UP
```
jobs [−l| −p][job_id...]
```

DESCRIPTION

The *jobs* utility shall display the status of jobs that were started in the current shell environment; see Section 2.12 (on page 2263).

When *jobs* reports the termination status of a job, the shell shall remove its process ID from the list of those ‘‘known in the current shell execution environment’’; see Section 2.9.3.1 (on page 2252).

OPTIONS

The *jobs* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−l	(The letter ell.) Provide more information about each job listed. This information shall include the job number, current job, process group ID, state, and the command that formed the job.
−p	Display only the process IDs for the process group leaders of the selected jobs.

By default, the *jobs* utility shall display the status of all stopped jobs, running background jobs and all jobs whose status has changed and have not been reported by the shell.

OPERANDS

The following operand shall be supported:

job_id	Specifies the jobs for which the status is to be displayed. If no *job_id* is given, the status information for all jobs shall be displayed. The format of *job_id* is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.203, Job Control Job ID.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *jobs*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of

diagnostic messages written to standard error and informative messages written to standard output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the **–p** option is specified, the output shall consist of one line for each process ID:

`"%d\n", <process ID>`

Otherwise, if the **–l** option is not specified, the output shall be a series of lines of the form:

`"[%d] %c %s %s\n", <job-number>, <current>, <state>, <command>`

where the fields shall be as follows:

<*current*> The character `'+'` identifies the job that would be used as a default for the *fg* or *bg* utilities; this job can also be specified using the *job_id* %+ or `"%%"`. The character `'–'` identifies the job that would become the default if the current default job were to exit; this job can also be specified using the *job_id* %–. For other jobs, this field is a <space>. At most one job can be identified with `'+'` and at most one job can be identified with `'–'`. If there is any suspended job, then the current job shall be a suspended job. If there are at least two suspended jobs, then the previous job also shall be a suspended job.

<*job-number*> A number that can be used to identify the process group to the *wait*, *fg*, *bg*, and *kill* utilities. Using these utilities, the job can be identified by prefixing the job number with `'%'`.

<*state*> One of the following strings (in the POSIX locale):

Running Indicates that the job has not been suspended by a signal and has not exited.

Done Indicates that the job completed and returned exit status zero.

Done(*code*) Indicates that the job completed normally and that it exited with the specified non-zero exit status, *code*, expressed as a decimal number.

Stopped Indicates that the job was suspended by the SIGTSTP signal.

Stopped (**SIGTSTP**)
Indicates that the job was suspended by the SIGTSTP signal.

Stopped (**SIGSTOP**)
Indicates that the job was suspended by the SIGSTOP signal.

Stopped (**SIGTTIN**)
Indicates that the job was suspended by the SIGTTIN signal.

Stopped (**SIGTTOU**)
Indicates that the job was suspended by the SIGTTOU signal.

The implementation may substitute the string **Suspended** in place of **Stopped**. If the job was terminated by a signal, the format of <*state*> is unspecified, but it shall be visibly distinct from all of the other <*state*> formats shown here and shall indicate the name or description of the signal causing the termination.

<*command*> The associated command that was given to the shell.

If the **–l** option is specified, a field containing the process group ID shall be inserted before the <*state*> field. Also, more processes in a process group may be output on separate lines, using only the process ID and <*command*> fields.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The **–p** option is the only portable way to find out the process group of a job because different implementations have different strategies for defining the process group of the job. Usage such as $(*jobs* **–p**) provides a way of referring to the process group of the job in an implementation-independent way.

The *jobs* utility does not work as expected when it is operating in its own utility execution environment because that environment has no applicable jobs to manipulate. See the APPLICATION USAGE section for *bg* (on page 2410). For this reason, *jobs* is generally implemented as a shell regular built-in.

EXAMPLES

None.

RATIONALE

Both "%%" and "%+" are used to refer to the current job. Both forms are of equal validity—the "%%" mirroring "$$" and "%+" mirroring the output of *jobs*. Both forms reflect historical practice of the KornShell and the C shell with job control.

The job control features provided by *bg*, *fg*, and *jobs* are based on the KornShell. The standard developers examined the characteristics of the C shell versions of these utilities and found that differences exist. Despite widespread use of the C shell, the KornShell versions were selected for this volume of IEEE Std 1003.1-200x to maintain a degree of uniformity with the rest of the KornShell features selected (such as the very popular command line editing features).

The *jobs* utility is not dependent on the job control option, as are the seemingly related *bg* and *fg* utilities because *jobs* is useful for examining background jobs, regardless of the condition of job control. When the user has invoked a *set* **+m** command and job control has been turned off, *jobs* can still be used to examine the background jobs associated with that current session. Similarly, *kill* can then be used to kill background jobs with *kill*% <*background job number*>.

The output for terminated jobs is left unspecified to accommodate various historical systems. The following formats have been witnessed:

1. **Killed**(*signal name*)
2. *signal name*
3. *signal name*(**coredump**)
4. *signal description* – **core dumped**

Most users should be able to understand these formats, although it means that applications have trouble parsing them.

The calculation of job IDs was not described since this would suggest an implementation, which may impose unnecessary restrictions.

In an early proposal, a –**n** option was included to ‘‘Display the status of jobs that have changed, exited, or stopped since the last status report’’. It was removed because the shell always writes any changed status of jobs before each prompt.

FUTURE DIRECTIONS

None.

SEE ALSO

bg, *fg*, *kill*, *wait*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The JC shading is removed as job control is mandatory in this issue.

NAME

join — relational database operator

SYNOPSIS

```
join [−a file_number | −v file_number][−e string][−o list][−t char]
    [−1 field][−2 field] file1 file2
```

DESCRIPTION

The *join* utility shall perform an equality join on the files *file1* and *file2*. The joined files shall be written to the standard output.

The join field is a field in each file on which the files are compared. The *join* utility shall write one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line by default shall consist of the join field, then the remaining fields from *file1*, then the remaining fields from *file2*. This format can be changed by using the **−o** option (see below). The **−a** option can be used to add unmatched lines to the output. The **−v** option can be used to output only unmatched lines.

The files *file1* and *file2* shall be ordered in the collating sequence of *sort* **−b** on the fields on which they shall be joined, by default the first in each line. All selected output shall be written in the same collating sequence.

The default input field separators shall be <blank>s. In this case, multiple separators shall count as one field separator, and leading separators shall be ignored. The default output field separator shall be a <space>.

The field separator and collating sequence can be changed by using the **−t** option (see below).

If the same key appears more than once in either file, all combinations of the set of remaining fields in *file1* and the set of remaining fields in *file2* are output in the order of the lines encountered.

If the input files are not in the appropriate collating sequence, the results are unspecified.

OPTIONS

The *join* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a *file_number*
Produce a line for each unpairable line in file *file_number*, where *file_number* is 1 or 2, in addition to the default output. If both **−a**1 and **−a**2 are specified, all unpairable lines shall be output.

−e *string* Replace empty output fields in the list selected by **−o** with the string *string*.

−o *list* Construct the output line to comprise the fields specified in *list*, each element of which shall have one of the following two forms:

1. *file_number.field*, where *file_number* is a file number and *field* is a decimal integer field number
2. 0 (zero), representing the join field

The elements of *list* shall be either comma-separated or <blank>-separated, as specified in Guideline 8 of the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines. The fields specified by *list* shall be written for all selected output lines. Fields selected by *list* that do not appear in the input shall be treated as empty output fields. (See the **−e** option.) Only specifically requested fields shall be written. The application shall ensure that *list* is a single command line argument.

−t *char* Use character *char* as a separator, for both input and output. Every appearance of *char* in a line shall be significant. When this option is specified, the collating sequence shall be the same as *sort* without the **−b** option. |

−v *file_number*
Instead of the default output, produce a line only for each unpairable line in *file_number*, where *file_number* is 1 or 2. If both **−v**1 and **−v**2 are specified, all unpairable lines shall be output.

−1 *field* Join on the *field*th field of file 1. Fields are decimal integers starting with 1.

−2 *field* Join on the *field*th field of file 2. Fields are decimal integers starting with 1.

OPERANDS

The following operands shall be supported:

file1, *file2*
A pathname of a file to be joined. If either of the *file1* or *file2* operands is `'−'`, the standard input shall be used in its place.

STDIN

The standard input shall be used only if the *file1* or *file2* operand is `'−'`. See the INPUT FILES section.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *join*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE
Determine the locale of the collating sequence *join* expects to have been used when the input files were sorted.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *join* utility output shall be a concatenation of selected character fields. When the −**o** option is not specified, the output shall be:

```
"%s%s%s\n", <join field>, <other file1 fields>,
    <other file2 fields>
```

If the join field is not the first field in a file, the <*other file fields*> for that file shall be:

```
<fields preceding join field>, <fields following join field>
```

When the −**o** option is specified, the output format shall be:

```
"%s\n", <concatenation of fields>
```

where the concatenation of fields is described by the −**o** option, above.

For either format, each field (except the last) shall be written with its trailing separator character. If the separator is the default (<blank>s), a single <space> shall be written after each field (except the last).

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were output successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Pathnames consisting of numeric digits or of the form *string.string* should not be specified directly following the −**o** list.

EXAMPLES

The −**o** 0 field essentially selects the union of the join fields. For example, given file **phone**:

```
!Name           Phone Number
Don             +1 123-456-7890
Hal             +1 234-567-8901
Yasushi         +2 345-678-9012
```

and file **fax**:

```
!Name           Fax Number
Don             +1 123-456-7899
Keith           +1 456-789-0122
Yasushi         +2 345-678-9011
```

(where the large expanses of white space are meant to each represent a single <tab>), the command:

```
join −t "<tab>" −a 1 −a 2 −e ’(unknown)’ −o 0,1.2,2.2 phone fax
```

would produce:

```
!Name           Phone Number            Fax Number
Don             +1 123-456-7890         +1 123-456-7899
Hal             +1 234-567-8901         (unknown)
Keith           (unknown)               +1 456-789-0122
Yasushi         +2 345-678-9012         +2 345-678-9011
```

Multiple instances of the same key will produce combinatorial results. The following:

```
fa:
    a x
    a y
    a z
fb:
    a p
```

will produce:

```
a x p
a y p
a z p
```

And the following:

```
fa:
    a b c
    a d e
fb:
    a w x
    a y z
    a o p
```

will produce:

```
a b c w x
a b c y z
a b c o p
a d e w x
a d e y z
a d e o p
```

RATIONALE

The −**e** option is only effective when used with −**o** because, unless specific fields are identified using −**o**, *join* is not aware of what fields might be empty. The exception to this is the join field, but identifying an empty join field with the −**e** string is not historical practice and some scripts might break if this were changed.

The 0 field in the −**o** list was adopted from the Tenth Edition version of *join* to satisfy international objections that the *join* in the base documents do not support the ‘‘full join’’ or ‘‘outer join’’ described in relational database literature. Although it has been possible to include a join field in the output (by default, or by field number using −**o**), the join field could not be included for an unpaired line selected by −**a**. The −**o** 0 field essentially selects the union of the join fields.

This sort of outer join was not possible with the *join* commands in the base documents. The −**o** 0 field was chosen because it is an upward-compatible change for applications. An alternative was considered: have the join field represent the union of the fields in the files (where they are identical for matched lines, and one or both are null for unmatched lines). This was not adopted because it would break some historical applications.

The ability to specify *file2* as – is not historical practice; it was added for completeness.

The –**v** option is not historical practice, but was considered necessary because it permitted the writing of *only* those lines that do not match on the join field, as opposed to the –**a** option, which prints both lines that do and do not match. This additional facility is parallel with the –**v** option of *grep*.

Some historical implementations have been encountered where a blank line in one of the input files was considered to be the end of the file; the description in this volume of IEEE Std 1003.1-200x does not cite this as an allowable case.

FUTURE DIRECTIONS

None.

SEE ALSO

awk, *comm*, *sort*, *uniq*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The obsolescent –**j** options and the multi-argument –**o** option are withdrawn in this issue.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

kill — terminate or signal processes

SYNOPSIS

```
kill -s signal_name pid ...
kill -l [exit_status]
XSI kill [-signal_name] pid ...
kill [-signal_number] pid ...
```

DESCRIPTION

The *kill* utility shall send a signal to the process or processes specified by each *pid* operand.

For each *pid* operand, the *kill* utility shall perform actions equivalent to the *kill*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x called with the following arguments:

- The value of the *pid* operand shall be used as the *pid* argument.
- The *sig* argument is the value specified by the **–s** option, –*signal_number* option, or the –*signal_name* option, or by SIGTERM, if none of these options is specified.

OPTIONS

The *kill* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, XSI except that in the last two SYMOPSIS forms, the –*signal_number* and –*signal_name* options are usually more than a single character.

The following options shall be supported:

–l (The letter ell.) Write all values of *signal_name* supported by the implementation, if no operand is given. If an *exit_status* operand is given and it is a value of the `'?'` shell special parameter (see Section 2.5.2 (on page 2235) and *wait* (on page 3239)) corresponding to a process that was terminated by a signal, the *signal_name* corresponding to the signal that terminated the process shall be written. If an *exit_status* operand is given and it is the unsigned decimal integer value of a signal number, the *signal_name* (the symbolic constant name without the **SIG** prefix defined in the Base Definitions volume of IEEE Std 1003.1-200x) corresponding to that signal shall be written. Otherwise, the results are unspecified.

–s *signal_name*

Specify the signal to send, using one of the symbolic names defined in the **<signal.h>** header. Values of *signal_name* shall be recognized in a case-independent fashion, without the **SIG** prefix. In addition, the symbolic name 0 shall be recognized, representing the signal value zero. The corresponding signal shall be sent instead of SIGTERM.

XSI –*signal_name*

Equivalent to –**s** *signal_name*.

XSI –*signal_number*

Specify a non-negative decimal integer, *signal_number*, representing the signal to be used instead of SIGTERM, as the *sig* argument in the effective call to *kill*( ). The correspondence between integer values and the *sig* value used is shown in the following table.

The effects of specifying any *signal_number* other than those listed in the table are undefined.

signal_number	*sig* **Value**
0	0
1	SIGHUP
2	SIGINT
3	SIGQUIT
6	SIGABRT
9	SIGKILL
14	SIGALRM
15	SIGTERM

XSI

If the first argument is a negative integer, it shall be interpreted as a −*signal_number* option, not as a negative *pid* operand specifying a process group.

OPERANDS

The following operands shall be supported:

pid One of the following:

1. A decimal integer specifying a process or process group to be signaled. The process or processes selected by positive, negative and zero values of the *pid* operand shall be as described for the *kill*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. If process number 0 is specified, all processes in the current process group shall be signaled. For the effects of negative *pid* numbers, see the *kill*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. If the first *pid* operand is negative, it should be preceded by "−−" to keep it from being interpreted as an option.
2. A job control job ID (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.203, Job Control Job ID) that identifies a background process group to be signaled. The job control job ID notation is applicable only for invocations of *kill* in the current shell execution environment; see Section 2.12 (on page 2263).

exit_status A decimal integer specifying a signal number or the exit status of a process terminated by a signal.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *kill*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–l** option is not specified, the standard output shall not be used.

When the **–l** option is specified, the symbolic name of each signal shall be written in the following format:

```
"%s%c", <signal_name>, <separator>
```

where the <*signal_name*> is in uppercase, without the **SIG** prefix, and the <*separator*> shall be either a <newline> or a <space>. For the last signal written, <*separator*> shall be a <newline>.

When both the **–l** option and *exit_status* operand are specified, the symbolic name of the corresponding signal shall be written in the following format:

```
"%s\n", <signal_name>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	At least one matching process was found for each *pid* operand, and the specified signal was successfully processed for at least one matching process.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Process numbers can be found by using *ps*.

The job control job ID notation is not required to work as expected when *kill* is operating in its own utility execution environment. In either of the following examples:

```
nohup kill %1 &
system("kill %1");
```

the *kill* operates in a different environment and does not share the shell's understanding of job numbers.

EXAMPLES

Any of the commands:

```
kill –9 100 –165
kill –s kill 100 –165
kill –s KILL 100 –165
```

sends the SIGKILL signal to the process whose process ID is 100 and to all processes whose process group ID is 165, assuming the sending process has permission to send that signal to the specified processes, and that they exist.

The System Interfaces volume of IEEE Std 1003.1-200x and this volume of IEEE Std 1003.1-200x do not require specific signal numbers for any *signal_names*. Even the −*signal_number* option provides symbolic (although numeric) names for signals. If a process is terminated by a signal, its exit status indicates the signal that killed it, but the exact values are not specified. The *kill* −**l** option, however, can be used to map decimal signal numbers and exit status values into the name of a signal. The following example reports the status of a terminated job:

```
job
stat=$?
if [ $stat −eq 0 ]
then
    echo job completed successfully.
elif [ $stat −gt 128 ]
then
    echo job terminated by signal SIG$(kill −l $stat).
else
    echo job terminated with error code $stat.
fi
```

To send the default signal to a process group (say 123), an application should use a command similar to one of the following: |

```
kill −TERM −123
kill −− −123
```

RATIONALE

The −**l** option originated from the C shell, and is also implemented in the KornShell. The C shell output can consist of multiple output lines because the signal names do not always fit on a single line on some terminal screens. The KornShell output also included the implementation-defined signal numbers and was considered by the standard developers to be too difficult for scripts to parse conveniently. The specified output format is intended not only to accommodate the historical C shell output, but also to permit an entirely vertical or entirely horizontal listing on systems for which this is appropriate.

An early proposal invented the name SIGNULL as a *signal_name* for signal 0 (used by the System Interfaces volume of IEEE Std 1003.1-200x to test for the existence of a process without sending it a signal). Since the *signal_name* 0 can be used in this case unambiguously, SIGNULL has been removed.

An early proposal also required symbolic *signal_names* to be recognized with or without the **SIG** prefix. Historical versions of *kill* have not written the **SIG** prefix for the −**l** option and have not recognized the **SIG** prefix on *signal_names*. Since neither applications portability nor ease-of-use would be improved by requiring this extension, it is no longer required.

To avoid an ambiguity of an initial negative number argument specifying either a signal number or a process group, IEEE Std 1003.1-200x mandates that it is always considered the former by implementations that support the XSI option. It also requires that conforming applications always use the "−−" options terminator argument when specifying a process group, unless an option is also specified. |

The −**s** option was added in response to international interest in providing some form of *kill* that meets the Utility Syntax Guidelines. |

The job control job ID notation is not required to work as expected when *kill* is operating in its own utility execution environment. In either of the following examples:

```
nohup kill %1 &
system("kill %1");
```

the *kill* operates in a different environment and does not understand how the shell has managed its job numbers.

FUTURE DIRECTIONS

None.

SEE ALSO

ps, *wait*, the System Interfaces volume of IEEE Std 1003.1-200x, *kill*( ), the Base Definitions volume of IEEE Std 1003.1-200x, **<signal.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

The obsolescent versions of the SYNOPSIS were turned into non-obsolescent features of the XSI option, corresponding to a similar change in the *trap* special built-in.

NAME

lex — generate programs for lexical tasks (**DEVELOPMENT**)

SYNOPSIS

CD
```
lex [−t][−n|−v][file ...]
```

DESCRIPTION

The *lex* utility shall generate C programs to be used in lexical processing of character input, and that can be used as an interface to *yacc*. The C programs shall be generated from *lex* source code and conform to the ISO C standard. Usually, the *lex* utility shall write the program it generates to the file **lex.yy.c**; the state of this file is unspecified if *lex* exits with a non-zero exit status. See the EXTENDED DESCRIPTION section for a complete description of the *lex* input language.

OPTIONS

The *lex* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−n	Suppress the summary of statistics usually written with the **−v** option. If no table sizes are specified in the *lex* source code and the **−v** option is not specified, then **−n** is implied.
−t	Write the resulting program to standard output instead of **lex.yy.c**.
−v	Write a summary of *lex* statistics to the standard output. (See the discussion of *lex* table sizes in **Definitions in lex** (on page 2736).) If the **−t** option is specified and **−n** is not specified, this report shall be written to standard error. If table sizes are specified in the *lex* source code, and if the **−n** option is not specified, the **−v** option may be enabled.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If more than one such *file* is specified, all files shall be concatenated to produce a single *lex* program. If no *file* operands are specified, or if a *file* operand is `'−'`, the standard input shall be used.

STDIN

The standard input shall be used if no *file* operands are specified, or if a *file* operand is `'−'`. See INPUT FILES.

INPUT FILES

The input files shall be text files containing *lex* source code, as described in the EXTENDED DESCRIPTION section.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *lex*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-

character collating elements within regular expressions. If this variable is not set to the POSIX locale, the results are unspecified.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), and the behavior of character classes within regular expressions. If this variable is not set to the POSIX locale, the results are unspecified.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the **–t** option is specified, the text file of C source code output of *lex* shall be written to standard output.

If the **–t** option is not specified:

- Implementation-defined informational, error, and warning messages concerning the contents of *lex* source code input shall be written to either the standard output or standard error.
- If the **–v** option is specified and the **–n** option is not specified, *lex* statistics shall also be written to either the standard output or standard error, in an implementation-defined format. These statistics may also be generated if table sizes are specified with a '%' operator in the *Definitions* section, as long as the **–n** option is not specified.

STDERR

If the **–t** option is specified, implementation-defined informational, error, and warning messages concerning the contents of *lex* source code input shall be written to the standard error.

If the **–t** option is not specified:

1. Implementation-defined informational, error, and warning messages concerning the contents of *lex* source code input shall be written to either the standard output or standard error.
2. If the **–v** option is specified and the **–n** option is not specified, *lex* statistics shall also be written to either the standard output or standard error, in an implementation-defined format. These statistics may also be generated if table sizes are specified with a '%' operator in the *Definitions* section, as long as the **–n** option is not specified.

OUTPUT FILES

A text file containing C source code shall be written to **lex.yy.c**, or to the standard output if the **–t** option is present.

EXTENDED DESCRIPTION

Each input file shall contain *lex* source code, which is a table of regular expressions with corresponding actions in the form of C program fragments. |

When **lex.yy.c** is compiled and linked with the *lex* library (using the **–l l** operand with *c99*), the resulting program shall read character input from the standard input and shall partition it into strings that match the given expressions. |

When an expression is matched, these actions shall occur:

- The input string that was matched shall be left in *yytext* as a null-terminated string; *yytext* shall either be an external character array or a pointer to a character string. As explained in **Definitions in lex**, the type can be explicitly selected using the **%array** or **%pointer** declarations, but the default is implementation-defined.
- The external **int** *yyleng* shall be set to the length of the matching string.
- The expression's corresponding program fragment, or action, shall be executed.

During pattern matching, *lex* shall search the set of patterns for the single longest possible match. Among rules that match the same number of characters, the rule given first shall be chosen.

The general format of *lex* source shall be:

Definitions
%%
Rules
%%
*User*Subroutines

The first `"%%"` is required to mark the beginning of the rules (regular expressions and actions); the second `"%%"` is required only if user subroutines follow.

Any line in the *Definitions* section beginning with a <blank> shall be assumed to be a C program fragment and shall be copied to the external definition area of the **lex.yy.c** file. Similarly, anything in the *Definitions* section included between delimiter lines containing only `"%{"` and `"%}"` shall also be copied unchanged to the external definition area of the **lex.yy.c** file.

Any such input (beginning with a <blank> or within `"%{"` and `"%}"` delimiter lines) appearing at the beginning of the *Rules* section before any rules are specified shall be written to **lex.yy.c** after the declarations of variables for the *yylex*( ) function and before the first line of code in *yylex*( ). Thus, user variables local to *yylex*( ) can be declared here, as well as application code to execute upon entry to *yylex*( ).

The action taken by *lex* when encountering any input beginning with a <blank> or within `"%{"` and `"%}"` delimiter lines appearing in the *Rules* section but coming after one or more rules is undefined. The presence of such input may result in an erroneous definition of the *yylex*( ) function.

Definitions in lex

Definitions appear before the first `"%%"` delimiter. Any line in this section not contained between `"%{"` and `"%}"` lines and not beginning with a <blank> shall be assumed to define a *lex* substitution string. The format of these lines shall be:

```
name substitute
```

If a *name* does not meet the requirements for identifiers in the ISO C standard, the result is undefined. The string *substitute* shall replace the string {*name*} when it is used in a rule. The *name* string shall be recognized in this context only when the braces are provided and when it does not appear within a bracket expression or within double-quotes.

In the *Definitions* section, any line beginning with a `'%'` (percent sign) character and followed by an alphanumeric word beginning with either `'s'` or `'S'` shall define a set of start conditions. Any line beginning with a `'%'` followed by a word beginning with either `'x'` or `'X'` shall define a set of exclusive start conditions. When the generated scanner is in a `"%s"` state, patterns with no state specified shall be also active; in a "%x" state, such patterns shall not be active. The rest of the line, after the first word, shall be considered to be one or more <blank>-separated names of start conditions. Start condition names shall be constructed in the same way as definition names. Start conditions can be used to restrict the matching of regular expressions to one or more states as described in **Regular Expressions in lex** (on page 2738).

Implementations shall accept either of the following two mutually exclusive declarations in the *Definitions* section:

%array Declare the type of *yytext* to be a null-terminated character array.

%pointer Declare the type of *yytext* to be a pointer to a null-terminated character string.

The default type of *yytext* is implementation-defined. If an application refers to *yytext* outside of the scanner source file (that is, via an **extern**), the application shall include the appropriate **%array** or **%pointer** declaration in the scanner source file.

Implementations shall accept declarations in the *Definitions* section for setting certain internal table sizes. The declarations are shown in the following table.

Table 4-9 Table Size Declarations in *lex*

Declaration	Description	Minimum Value
%**p** *n*	Number of positions	2 500
%**n** *n*	Number of states	500
%**a** *n*	Number of transitions	2 000
%**e** *n*	Number of parse tree nodes	1 000
%**k** *n*	Number of packed character classes	1 000
%**o** *n*	Size of the output array	3 000

In the table, *n* represents a positive decimal integer, preceded by one or more <blank>s. The exact meaning of these table size numbers is implementation-defined. The implementation shall document how these numbers affect the *lex* utility and how they are related to any output that may be generated by the implementation should limitations be encountered during the execution of *lex*. It shall be possible to determine from this output which of the table size values needs to be modified to permit *lex* to successfully generate tables for the input language. The values in the column Minimum Value represent the lowest values conforming implementations shall provide.

Rules in lex

The rules in *lex* source files are a table in which the left column contains regular expressions and the right column contains actions (C program fragments) to be executed when the expressions are recognized.

```
ERE action
ERE action
...
```

The extended regular expression (*ERE*) portion of a row shall be separated from *action* by one or more <blank>s. A regular expression containing <blank>s shall be recognized under one of the following conditions:

- The entire expression appears within double-quotes.
- The <blank>s appear within double-quotes or square brackets.

- Each <blank> is preceded by a backslash character.

User Subroutines in lex

Anything in the user subroutines section shall be copied to **lex.yy.c** following *yylex*( ).

Regular Expressions in lex

The *lex* utility shall support the set of extended regular expressions (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.4, Extended Regular Expressions), with the following additions and exceptions to the syntax:

`"..."` Any string enclosed in double-quotes shall represent the characters within the double-quotes as themselves, except that backslash escapes (which appear in the following table) shall be recognized. Any backslash-escape sequence shall be terminated by the closing quote. For example, `"\01""1"` represents a single string: the octal value 1 followed by the character `'1'`.

<*state*>*r*, <*state1*,*state2*,. . .>*r*

The regular expression *r* shall be matched only when the program is in one of the start conditions indicated by *state*, *state1*, and so on; see **Actions in lex** (on page 2740). (As an exception to the typographical conventions of the rest of this volume of IEEE Std 1003.1-200x, in this case <*state*> does not represent a metavariable, but the literal angle-bracket characters surrounding a symbol.) The start condition shall be recognized as such only at the beginning of a regular expression.

r/*x* The regular expression *r* shall be matched only if it is followed by an occurrence of regular expression *x* (*x* is the instance of trailing context, further defined below). The token returned in *yytext* shall only match *r*. If the trailing portion of *r* matches the beginning of *x*, the result is unspecified. The *r* expression cannot include further trailing context or the `'$'` (match-end-of-line) operator; *x* cannot include the `'^'` (match-beginning-of-line) operator, nor trailing context, nor the `'$'` operator. That is, only one occurrence of trailing context is allowed in a *lex* regular expression, and the `'^'` operator only can be used at the beginning of such an expression.

{*name*} When *name* is one of the substitution symbols from the *Definitions* section, the string, including the enclosing braces, shall be replaced by the *substitute* value. The *substitute* value shall be treated in the extended regular expression as if it were enclosed in parentheses. No substitution shall occur if {*name*} occurs within a bracket expression or within double-quotes.

Within an ERE, a backslash character shall be considered to begin an escape sequence as specified in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`). In addition, the escape sequences in the following table shall be recognized.

A literal <newline> cannot occur within an ERE; the escape sequence `'\n'` can be used to represent a <newline>. A <newline> shall not be matched by a period operator.

Table 4-10 Escape Sequences in *lex*

Escape Sequence	Description	Meaning
`\digits`	A backslash character followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0 (that is, representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the one, two, or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-defined. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading `'\'` for each byte.
`\xdigits`	A backslash character followed by the longest sequence of hexadecimal-digit characters (01234567abcdefABCDEF). If all of the digits are 0 (that is, representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the hexadecimal integer.
`\c`	A backslash character followed by any character not described in this table or in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`).	The character `'c'`, unchanged.

Note: If a `'\x'` sequence needs to be immediately followed by a hexadecimal digit character, a sequence such as `"\x1""1"` can be used, which represents a character containing the value 1, followed by the character `'1'`.

The order of precedence given to extended regular expressions for *lex* differs from that specified in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.4, Extended Regular Expressions. The order of precedence for *lex* shall be as shown in the following table, from high to low.

Note: The escaped characters entry is not meant to imply that these are operators, but they are included in the table to show their relationships to the true operators. The start condition, trailing context, and anchoring notations have been omitted from the table because of the placement restrictions described in this section; they can only appear at the beginning or ending of an ERE.

Table 4-11 ERE Precedence in *lex*

Extended Regular Expression	Precedence
collation-related bracket symbols	`[= =]  [: :]  [. .]`
escaped characters	`\<`*special character*`>`
bracket expression	`[ ]`
quoting	`"..."`
grouping	`( )`
definition	`{`*name*`}`
single-character RE duplication	`* + ?`
concatenation	
interval expression	`{m,n}`
alternation	`|`

The ERE anchoring operators `'^'` and `'$'`) do not appear in the table. With *lex* regular expressions, these operators are restricted in their use: the `'^'` operator can only be used at the beginning of an entire regular expression, and the `'$'` operator only at the end. The operators apply to the entire regular expression. Thus, for example, the pattern `"(^abc)|(def$)"` is undefined; it can instead be written as two separate rules, one with the regular expression `"^abc"` and one with `"def$"`, which share a common action via the special `'|'` action (see below). If the pattern were written `"^abc|def$"`, it would match either `"abc"` or `"def"` on a line by itself.

Unlike the general ERE rules, embedded anchoring is not allowed by most historical *lex* implementations. An example of embedded anchoring would be for patterns such as `"(^| )foo( |$)"` to match `"foo"` when it exists as a complete word. This functionality can be obtained using existing *lex* features:

```
^foo/[ \n]      |
" foo"/[ \n]    /* Found foo as a separate word. */
```

Note also that `'$'` is a form of trailing context (it is equivalent to `"/\n"`) and as such cannot be used with regular expressions containing another instance of the operator (see the preceding discussion of trailing context).

The additional regular expressions trailing-context operator `'/'` can be used as an ordinary character if presented within double-quotes, `"/"`; preceded by a backslash, `"\/"`; or within a bracket expression, `"[/]"`. The start-condition `'<'` and `'>'` operators shall be special only in a start condition at the beginning of a regular expression; elsewhere in the regular expression they shall be treated as ordinary characters.

Actions in lex

The action to be taken when an ERE is matched can be a C program fragment or the special actions described below; the program fragment can contain one or more C statements, and can also include special actions. The empty C statement `';'` shall be a valid action; any string in the **lex.yy.c** input that matches the pattern portion of such a rule is effectively ignored or skipped. However, the absence of an action shall not be valid, and the action *lex* takes in such a condition is undefined.

The specification for an action, including C statements and special actions, can extend across several lines if enclosed in braces:

```
ERE <one or more blanks> { program statement
                           program statement }
```

The default action when a string in the input to a **lex.yy.c** program is not matched by any expression shall be to copy the string to the output. Because the default behavior of a program generated by *lex* is to read the input and copy it to the output, a minimal *lex* source program that has just `"%%"` shall generate a C program that simply copies the input to the output unchanged.

Four special actions shall be available:

`|    ECHO;    REJECT;    BEGIN`

`|` The action `'|'` means that the action for the next rule is the action for this rule. Unlike the other three actions, `'|'` cannot be enclosed in braces or be semicolon-terminated; the application shall ensure that it is specified alone, with no other actions.

ECHO; Write the contents of the string *yytext* on the output.

REJECT; Usually only a single expression is matched by a given string in the input. **REJECT** means "continue to the next expression that matches the current input", and shall cause whatever rule was the second choice after the current rule to be executed for the same input. Thus, multiple rules can be matched and executed for one input string or overlapping input strings. For example, given the regular expressions `"xyz"` and `"xy"` and the input `"xyz"`, usually only the regular expression `"xyz"` would match. The next attempted match would start after **z.** If the last action in the `"xyz"` rule is **REJECT**, both this rule and the `"xy"` rule would be executed. The **REJECT** action may be implemented in such a fashion that flow of control does not continue after it, as if it were equivalent to a **goto** to another part of *yylex*( ). The use of **REJECT** may result in somewhat larger and slower scanners.

BEGIN The action:

```
BEGIN newstate;
```

switches the state (start condition) to *newstate*. If the string *newstate* has not been declared previously as a start condition in the *Definitions* section, the results are unspecified. The initial state is indicated by the digit `'0'` or the token **INITIAL**.

The functions or macros described below are accessible to user code included in the *lex* input. It is unspecified whether they appear in the C code output of *lex*, or are accessible only through the **–l l** operand to *c99* (the *lex* library).

int *yylex*(**void**)
Performs lexical analysis on the input; this is the primary function generated by the *lex* utility. The function shall return zero when the end of input is reached; otherwise, it shall return non-zero values (tokens) determined by the actions that are selected.

int *yymore*(**void**)
When called, indicates that when the next input string is recognized, it is to be appended to the current value of *yytext* rather than replacing it; the value in *yyleng* shall be adjusted accordingly.

int *yyless*(**int** *n*)
Retains *n* initial characters in *yytext*, NUL-terminated, and treats the remaining characters as if they had not been read; the value in *yyleng* shall be adjusted accordingly.

int *input*(**void**)
Returns the next character from the input, or zero on end-of-file. It shall obtain input from the stream pointer *yyin*, although possibly via an intermediate buffer. Thus, once scanning has begun, the effect of altering the value of *yyin* is undefined. The character read shall be removed from the input stream of the scanner without any processing by the scanner.

int *unput*(**int** *c*)

Returns the character `'c'` to the input; *yytext* and *yyleng* are undefined until the next expression is matched. The result of using *unput*( ) for more characters than have been input is unspecified.

The following functions shall appear only in the *lex* library accessible through the **–l l** operand; they can therefore be redefined by a conforming application:

int *yywrap*(**void**)

Called by *yylex*( ) at end-of-file; the default *yywrap*( ) shall always return 1. If the application requires *yylex*( ) to continue processing with another source of input, then the application can include a function *yywrap*( ), which associates another file with the external variable **FILE *** *yyin* and shall return a value of zero.

int *main*(**int** *argc*, **char** **argv*[ ])

Calls *yylex*( ) to perform lexical analysis, then exits. The user code can contain *main*( ) to perform application-specific operations, calling *yylex*( ) as applicable.

Except for *input*( ), *unput*( ), and *main*( ), all external and static names generated by *lex* shall begin with the prefix **yy** or **YY**.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Conforming applications are warned that in the *Rules* section, an *ERE* without an action is not acceptable, but need not be detected as erroneous by *lex*. This may result in compilation or runtime errors.

The purpose of *input*( ) is to take characters off the input stream and discard them as far as the lexical analysis is concerned. A common use is to discard the body of a comment once the beginning of a comment is recognized.

The *lex* utility is not fully internationalized in its treatment of regular expressions in the *lex* source code or generated lexical analyzer. It would seem desirable to have the lexical analyzer interpret the regular expressions given in the *lex* source according to the environment specified when the lexical analyzer is executed, but this is not possible with the current *lex* technology. Furthermore, the very nature of the lexical analyzers produced by *lex* must be closely tied to the lexical requirements of the input language being described, which is frequently locale-specific anyway. (For example, writing an analyzer that is used for French text is not automatically useful for processing other languages.)

EXAMPLES

The following is an example of a *lex* program that implements a rudimentary scanner for a Pascal-like syntax:

```
%{
/* Need this for the call to atof() below. */
#include <math.h>
/* Need this for printf(), fopen(), and stdin below. */
#include <stdio.h>
%}
```

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%

{DIGIT}+ {
    printf("An integer: %s (%d)\n", yytext,
        atoi(yytext));
    }

{DIGIT}+"."{DIGIT}*        {
    printf("A float: %s (%g)\n", yytext,
        atof(yytext));
    }

if|then|begin|end|procedure|function        {
    printf("A keyword: %s\n", yytext);
    }

{ID}    printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/"        printf("An operator: %s\n", yytext);

"{"[^}\n]*"}"    /* Eat up one-line comments. */

[ \t\n]+        /* Eat up white space. */

.  printf("Unrecognized character: %s\n", yytext);

%%

int main(int argc, char *argv[])
{
    ++argv, --argc;  /* Skip over program name. */
    if (argc > 0)
        yyin = fopen(argv[0], "r");
    else
        yyin = stdin;

    yylex();
}
```

RATIONALE

Even though the **−c** option and references to the C language are retained in this description, *lex* may be generalized to other languages, as was done at one time for EFL, the Extended FORTRAN Language. Since the *lex* input specification is essentially language-independent, versions of this utility could be written to produce Ada, Modula-2, or Pascal code, and there are known historical implementations that do so.

The current description of *lex* bypasses the issue of dealing with internationalized EREs in the *lex* source code or generated lexical analyzer. If it follows the model used by *awk* (the source code is assumed to be presented in the POSIX locale, but input and output are in the locale specified by the environment variables), then the tables in the lexical analyzer produced by *lex* would interpret EREs specified in the *lex* source in terms of the environment variables specified when *lex* was executed. The desired effect would be to have the lexical analyzer interpret the EREs given in the *lex* source according to the environment specified when the lexical analyzer is executed, but this is not possible with the current *lex* technology.

The description of octal and hexadecimal-digit escape sequences agrees with the ISO C standard use of escape sequences. See the RATIONALE for *ed* (on page 2537) for a discussion of bytes larger than 9 bits being represented by octal values. Hexadecimal values can represent larger bytes and multi-byte characters directly, using as many digits as required.

There is no detailed output format specification. The observed behavior of *lex* under four different historical implementations was that none of these implementations consistently reported the line numbers for error and warning messages. Furthermore, there was a desire that *lex* be allowed to output additional diagnostic messages. Leaving message formats unspecified avoids these formatting questions and problems with internationalization.

Although the `%x` specifier for *exclusive* start conditions is not historical practice, it is believed to be a minor change to historical implementations and greatly enhances the usability of *lex* programs since it permits an application to obtain the expected functionality with fewer statements.

The **%array** and **%pointer** declarations were added as a compromise between historical systems. The System V-based *lex* copies the matched text to a *yytext* array. The *flex* program, supported in BSD and GNU systems, uses a pointer. In the latter case, significant performance improvements are available for some scanners. Most historical programs should require no change in porting from one system to another because the string being referenced is null-terminated in both cases. (The method used by *flex* in its case is to null-terminate the token in place by remembering the character that used to come right after the token and replacing it before continuing on to the next scan.) Multi-file programs with external references to *yytext* outside the scanner source file should continue to operate on their historical systems, but would require one of the new declarations to be considered strictly portable.

The description of EREs avoids unnecessary duplication of ERE details because their meanings within a *lex* ERE are the same as that for the ERE in this volume of IEEE Std 1003.1-200x.

The reason for the undefined condition associated with text beginning with a <blank> or within `"%{"` and `"%}"` delimiter lines appearing in the *Rules* section is historical practice. Both the BSD and System V *lex* copy the indented (or enclosed) input in the *Rules* section (except at the beginning) to unreachable areas of the *yylex*( ) function (the code is written directly after a *break* statement). In some cases, the System V *lex* generates an error message or a syntax error, depending on the form of indented input.

The intention in breaking the list of functions into those that may appear in **lex.yy.c** *versus* those that only appear in **libl.a** is that only those functions in **libl.a** can be reliably redefined by a conforming application.

The descriptions of standard output and standard error are somewhat complicated because historical *lex* implementations chose to issue diagnostic messages to standard output (unless **–t** was given). This standard allows this behavior, but leaves an opening for the more expected behavior of using standard error for diagnostics. Also, the System V behavior of writing the statistics when any table sizes are given is allowed, while BSD-derived systems can avoid it. The programmer can always precisely obtain the desired results by using either the **–t** or **–n** options.

The OPERANDS section does not mention the use of – as a synonym for standard input; not all historical implementations support such usage for any of the *file* operands.

A description of the *translation table* was deleted from early proposals because of its relatively low usage in historical applications.

The change to the definition of the *input*( ) function that allows buffering of input presents the opportunity for major performance gains in some applications.

The following examples clarify the differences between *lex* regular expressions and regular expressions appearing elsewhere in this volume of IEEE Std 1003.1-200x. For regular expressions of the form `"r/x"`, the string matching *r* is always returned; confusion may arise when the beginning of *x* matches the trailing portion of *r*. For example, given the regular expression `"a*b/cc"` and the input `"aaabcc"`, *yytext* would contain the string `"aaab"` on this match. But given the regular expression `"x*/xy"` and the input `"xxxy"`, the token **xxx**, not **xx**, is returned by some implementations because **xxx** matches `"x*"`.

In the rule `"ab*/bc"`, the `"b*"` at the end of *r* extends *r*'s match into the beginning of the trailing context, so the result is unspecified. If this rule were `"ab/bc"`, however, the rule matches the text `"ab"` when it is followed by the text `"bc"`. In this latter case, the matching of *r* cannot extend into the beginning of *x*, so the result is specified.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *yacc*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the C-Language Development Utilities option.

The obsolescent –**c** option is withdrawn in this issue.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

link — call *link*( ) function

SYNOPSIS

XSI
```
link file1 file2
```

DESCRIPTION

The *link* utility shall perform the function call:

```
link(file1, file2);
```

A user may need appropriate privilege to invoke the *link* utility.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

file1	The pathname of an existing file.
file2	The pathname of the new directory entry to be created.

STDIN

Not used.

INPUT FILES

Not used.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *link*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ln, *unlink*, the System Interfaces volume of IEEE Std 1003.1-200x, *link*( )

CHANGE HISTORY

First released in Issue 5.

NAME

ln — link files

SYNOPSIS

```
ln [−fs] source_file target_file

ln [−fs] source_file ... target_dir
```

DESCRIPTION

In the first synopsis form, the *ln* utility shall create a new directory entry (link) at the destination path specified by the *target_file* operand. If the **−s** option is specified, a symbolic link shall be created for the file specified by the *source_file* operand. This first synopsis form shall be assumed when the final operand does not name an existing directory; if more than two operands are specified and the final is not an existing directory, an error shall result.

In the second synopsis form, the *ln* utility shall create a new directory entry (link), or if the **−s** option is specified a symbolic link, for each file specified by a *source_file* operand, at a *destination* path in the existing directory named by *target_dir*.

If the last operand specifies an existing file of a type not specified by the System Interfaces volume of IEEE Std 1003.1-200x, the behavior is implementation-defined.

The corresponding *destination* path for each *source_file* shall be the concatenation of the target directory pathname, a slash character, and the last pathname component of the *source_file*. The second synopsis form shall be assumed when the final operand names an existing directory.

For each *source_file*:

1. If the *destination* path exists:
   a. If the **−f** option is not specified, *ln* shall write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_files*.
   b. Actions shall be performed equivalent to the *unlink*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called using *destination* as the *path* argument. If this fails for any reason, *ln* shall write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_files*.
2. If the **−s** option is specified, *ln* shall create a symbolic link named by the *destination* path and containing as its pathname *source_file*. The *ln* utility shall do nothing more with *source_file* and shall go on to any remaining files.
3. If *source_file* is a symbolic link, actions shall be performed equivalent to the *link*( ) function using the object that *source_file* references as the *path1* argument and the destination path as the *path2* argument. The *ln* utility shall do nothing more with *source_file* and shall go on to any remaining files.
4. Actions shall be performed equivalent to the *link*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x using *source_file* as the *path1* argument, and the *destination* path as the *path2* argument.

OPTIONS

The *ln* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−f	Force existing *destination* pathnames to be removed to allow the link.

−s	Create symbolic links instead of hard links.

OPERANDS

The following operands shall be supported:

source_file	A pathname of a file to be linked. If the **−s** option is specified, no restrictions on the type of file or on its existence shall be made. If the **−s** option is not specified, whether a directory can be linked is implementation-defined.
target_file	The pathname of the new directory entry to be created.
target_dir	A pathname of an existing directory in which the new directory entries are created.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ln*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All the specified files were linked successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

Some historic versions of *ln* (including the one specified by the SVID, unlink the destination file, if it exists, by default. If the mode does not permit writing, these versions prompt for confirmation before attempting the unlink. In these versions the −**f** option causes *ln* not to attempt to prompt for confirmation.

This allows *ln* to succeed in creating links when the target file already exists, even if the file itself is not writable (although the directory must be). Early proposals specified this functionality.

This volume of IEEE Std 1003.1-200x does not allow the *ln* utility to unlink existing destination paths by default for the following reasons:

- The *ln* utility has historically been used to provide locking for shell applications, a usage that is incompatible with *ln* unlinking the destination path by default. There was no corresponding technical advantage to adding this functionality.
- This functionality gave *ln* the ability to destroy the link structure of files, which changes the historical behavior of *ln*.
- This functionality is easily replicated with a combination of *rm* and *ln*.
- It is not historical practice in many systems; BSD and BSD-derived systems do not support this behavior. Unfortunately, whichever behavior is selected can cause scripts written expecting the other behavior to fail.
- It is preferable that *ln* perform in the same manner as the *link*( ) function, which does not permit the target to exist already.

This volume of IEEE Std 1003.1-200x retains the −**f** option to provide support for shell scripts depending on the SVID semantics. It seems likely that shell scripts would not be written to handle prompting by *ln* and would therefore have specified the −**f** option.

The −**f** option is an undocumented feature of many historical versions of the *ln* utility, allowing linking to directories. These versions require modification.

Early proposals of this volume of IEEE Std 1003.1-200x also required an −**i** option, which behaved like the −**i** options in *cp* and *mv*, prompting for confirmation before unlinking existing files. This was not historical practice for the *ln* utility and has been omitted.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, *find*, *pax*, *rm*, the System Interfaces volume of IEEE Std 1003.1-200x, *link*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

The *ln* utility is updated to include symbolic link processing as defined in the IEEE P1003.2b draft standard.

NAME

locale — get locale-specific information

SYNOPSIS

```
locale [−a| −m]

locale [−ck] name...
```

DESCRIPTION

The *locale* utility shall write information about the current locale environment, or all public locales, to the standard output. For the purposes of this section, a *public locale* is one provided by the implementation that is accessible to the application.

When *locale* is invoked without any arguments, it shall summarize the current locale environment for each locale category as determined by the settings of the environment variables defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale.

When invoked with operands, it shall write values that have been assigned to the keywords in the locale categories, as follows:

- Specifying a keyword name shall select the named keyword and the category containing that keyword.
- Specifying a category name shall select the named category and all keywords in that category.

OPTIONS

The *locale* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Write information about all available public locales. The available locales shall include **POSIX**, representing the POSIX locale. The manner in which the implementation determines what other locales are available is implementation-defined.
−c	Write the names of selected locale categories; see the STDOUT section. The **−c** option increases readability when more than one category is selected (for example, via more than one keyword name or via a category name). It is valid both with and without the **−k** option.
−k	Write the names and values of selected keywords. The implementation may omit values for some keywords; see the OPERANDS section.
−m	Write names of available charmaps; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set.

OPERANDS

The following operand shall be supported:

name	The name of a locale category as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, the name of a keyword in a locale category, or the reserved name **charmap**. The named category or keyword shall be selected for output. If a single *name* represents both a locale category name and a keyword name in the current locale, the results are unspecified. Otherwise, both category and keyword names can be specified as *name* operands, in any sequence. It is implementation-defined whether any keyword values are written for the categories *LC_CTYPE* and *LC_COLLATE*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *locale*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

XSI The application shall ensure that the *LANG*, *LC_**, and *NLSPATH* environment variables specify the current locale environment to be written out; they shall be used if the −**a** option is not specified.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If *locale* is invoked without any options or operands, the names and values of the *LANG* and *LC_** environment variables described in this volume of IEEE Std 1003.1-200x shall be written to the standard output, one variable per line, with *LANG* first, and each line using the following format. Only those variables set in the environment and not overridden by *LC_ALL* shall be written using this format:

```
"%s=%s\n", <variable_name>, <value>
```

The names of those *LC_** variables associated with locale categories defined in this volume of IEEE Std 1003.1-200x that are not set in the environment or are overridden by *LC_ALL* shall be written in the following format:

```
"%s=\""%s\""\n", <variable_name>, <implied value>
```

The <*implied value*> shall be the name of the locale that has been selected for that category by the implementation, based on the values in *LANG* and *LC_ALL*, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

The <*value*> and <*implied value*> shown above shall be properly quoted for possible later reentry to the shell. The <*value*> shall not be quoted using double-quotes (so that it can be distinguished by the user from the <*implied value*> case, which always requires double-quotes).

The *LC_ALL* variable shall be written last, using the first format shown above. If it is not set, it shall be written as:

```
"LC_ALL=\n"
```

If any arguments are specified:

1. If the −**a** option is specified, the names of all the public locales shall be written, each in the following format:

   `"%s\n", <locale name>`

2. If the −**c** option is specified, the names of all selected categories shall be written, each in the following format:

   `"%s\n", <category name>`

   If keywords are also selected for writing (see following items), the category name output shall precede the keyword output for that category.

   If the −**c** option is not specified, the names of the categories shall not be written; only the keywords, as selected by the <*name*> operand, shall be written.

3. If the −**k** option is specified, the names and values of selected keywords shall be written. If a value is non-numeric, it shall be written in the following format:

   `"%s=\"%s\"\n", <keyword name>, <keyword value>`

   If the keyword was **charmap**, the name of the charmap (if any) that was specified via the *localedef* −**f** option when the locale was created shall be written, with the word **charmap** as <*keyword name*>.

   If a value is numeric, it shall be written in one of the following formats:

   `"%s=%d\n", <keyword name>, <keyword value>`

   `"%s=%c%o\n", <keyword name>, <escape character>, <keyword value>`

   `"%s=%cx%x\n", <keyword name>, <escape character>, <keyword value>`

   where the <*escape character*> is that identified by the **escape_char** keyword in the current locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3, Locale Definition.

   Compound keyword values (list entries) shall be separated in the output by semicolons. When included in keyword values, the semicolon, the double-quote, the backslash, and any control character shall be preceded (escaped) with the escape character.

4. If the −**k** option is not specified, selected keyword values shall be written, each in the following format:

   `"%s\n", <keyword value>`

   If the keyword was **charmap**, the name of the charmap (if any) that was specified via the *localedef* −**f** option when the locale was created shall be written.

5. If the −**m** option is specified, then a list of all available charmaps shall be written, each in the format:

   `"%s\n", <charmap>`

   where <*charmap*> is in a format suitable for use as the option-argument to the *localedef* −**f** option.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 All the requested information was found and output successfully.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If the *LANG* environment variable is not set or set to an empty value, or one of the *LC_** environment variables is set to an unrecognized value, the actual locales assumed (if any) are implementation-defined as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

Implementations are not required to write out the actual values for keywords in the categories *LC_CTYPE* and *LC_COLLATE*; however, they must write out the categories (allowing an application to determine, for example, which character classes are available).

EXAMPLES

In the following examples, the assumption is that locale environment variables are set as follows:

```
LANG=locale_x
LC_COLLATE=locale_y
```

The command *locale* would result in the following output:

```
LANG=locale_x
LC_CTYPE="locale_x"
LC_COLLATE=locale_y
LC_TIME="locale_x"
LC_NUMERIC="locale_x"
LC_MONETARY="locale_x"
LC_MESSAGES="locale_x"
LC_ALL=
```

The order of presentation of the categories is not specified by this volume of IEEE Std 1003.1-200x.

The command:

```
LC_ALL=POSIX locale −ck decimal_point
```

would produce:

```
LC_NUMERIC
decimal_point="."
```

The following command shows an application of *locale* to determine whether a user-supplied response is affirmative:

```
if printf "%s\n" "$response" | grep –Eq "$(locale yesexpr)"
then
    affirmative processing goes here
else
    non-affirmative processing goes here
fi
```

RATIONALE

The output for categories *LC_CTYPE* and *LC_COLLATE* has been made implementation-defined because there is a questionable value in having a shell script receive an entire array of characters. It is also difficult to return a logical collation description, short of returning a complete *localedef* source.

The –**m** option was included to allow applications to query for the existence of charmaps. The output is a list of the charmaps (implementation-supplied and user-supplied, if any) on the system.

The –**c** option was included for readability when more than one category is selected (for example, via more than one keyword name or via a category name). It is valid both with and without the –**k** option.

The **charmap** keyword, which returns the name of the charmap (if any) that was used when the current locale was created, was included to allow applications needing the information to retrieve it.

FUTURE DIRECTIONS

None.

SEE ALSO

localedef, the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3, Locale Definition

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

localedef — define locale environment

SYNOPSIS

`localedef [−c][−f charmap][−i sourcefile][−u code_set_name] name`

DESCRIPTION

The *localedef* utility shall convert source definitions for locale categories into a format usable by the functions and utilities whose operational behavior is determined by the setting of the locale environment variables defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale. It is implementation-defined whether users have the capability to create new locales, in addition to those supplied by the implementation. If the symbolic constant POSIX2_LOCALEDEF is defined, the system supports the creation of new locales. XSI On XSI-conformant systems, the symbolic constant POSIX2_LOCALEDEF shall be defined.

The utility shall read source definitions for one or more locale categories belonging to the same locale from the file named in the **−i** option (if specified) or from standard input.

The *name* operand identifies the target locale. The utility shall support the creation of *public*, or generally accessible locales, as well as *private*, or restricted-access locales. Implementations may restrict the capability to create or modify public locales to users with the appropriate privileges.

Each category source definition shall be identified by the corresponding environment variable name and terminated by an **END** *category-name* statement. The following categories shall be supported. In addition, the input may contain source for implementation-defined categories.

LC_CTYPE	Defines character classification and case conversion.
LC_COLLATE	Defines collation rules.
LC_MONETARY	Defines the format and symbols used in formatting of monetary information.
LC_NUMERIC	Defines the decimal delimiter, grouping, and grouping symbol for non-monetary numeric editing.
LC_TIME	Defines the format and content of date and time information.
LC_MESSAGES	Defines the format and values of affirmative and negative responses.

OPTIONS

The *localedef* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	Create permanent output even if warning messages have been issued.
−f *charmap*	Specify the pathname of a file containing a mapping of character symbols and collating element symbols to actual character encodings. The format of the *charmap* is described under the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.4, Character Set Description File. The application shall ensure that this option is specified if symbolic names (other than collating symbols defined in a **collating-symbol** keyword) are used. If the **−f** option is not present, an implementation-defined character mapping shall be used.

–i *inputfile* The pathname of a file containing the source definitions. If this option is not present, source definitions shall be read from standard input. The format of the *inputfile* is described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3, Locale Definition.

–u *code_set_name*
Specify the name of a codeset used as the target mapping of character symbols and collating element symbols whose encoding values are defined in terms of the ISO / IEC 10646-1: 2000 standard position constant values.

OPERANDS

The following operand shall be supported:

name Identifies the locale; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale for a description of the use of this name. If the name contains one or more slash characters, *name* shall be interpreted as a pathname where the created locale definitions shall be stored. If *name* does not contain any slash characters, the interpretation of the name is implementation-defined and the locale shall be public. This capability may be restricted to users with appropriate privileges. (As a consequence of specifying one *name*, although several categories can be processed in one execution, only categories belonging to the same locale can be processed.)

STDIN

Unless the **–i** option is specified, the standard input shall be a text file containing one or more locale category source definitions, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3, Locale Definition. When lines are continued using the escape character mechanism, there is no limit to the length of the accumulated continued line.

INPUT FILES

The character set mapping file specified as the *charmap* option-argument is described under the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.4, Character Set Description File. If a locale category source definition contains a **copy** statement, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, and the **copy** statement names a valid, existing locale, then *localedef* shall behave as if the source definition had contained a valid category source definition for the named locale.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *localedef*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE
(This variable has no affect on *localedef*; the POSIX locale is used for this category.)

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files). This variable has no affect on the processing of *localedef* input data; the POSIX locale is used for this purpose, regardless of the value of this variable.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The utility shall report all categories successfully processed, in an unspecified format.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The format of the created output is unspecified. If the *name* operand does not contain a slash, the existence of an output file for the locale is unspecified.

EXTENDED DESCRIPTION

When the **–u** option is used, the *code_set_name* option-argument shall be interpreted as an implementation-defined name of a codeset to which the ISO/IEC 10646-1:2000 standard position constant values shall be converted via an implementation-defined method. Both the ISO/IEC 10646-1:2000 standard position constant values and other formats (decimal, hexadecimal, or octal) shall be valid as encoding values within the *charmap* file. The codeset represented by the implementation-defined name can be any codeset that is supported by the implementation.

When conflicts occur between the *charmap* specification of <*code_set_name*>, <*mb_cur_max*>, or <*mb_cur_min*> and the implementation-defined interpretation of these respective items for the codeset represented by the **–u** option-argument *code_set_name*, the result is unspecified.

When conflicts occur between the *charmap* encoding values specified for symbolic names of characters of the portable character set and the implementation-defined assignment of character encoding values, the result is unspecified.

If a non-printable character in the *charmap* has a width specified that is not **–1**, *localedef* shall generate a warning.

EXIT STATUS

The following exit values shall be returned:

0	No errors occurred and the locales were successfully created.
1	Warnings occurred and the locales were successfully created.
2	The locale specification exceeded implementation limits or the coded character set or sets used were not supported by the implementation, and no locale was created.
3	The capability to create new locales is not supported by the implementation.
>3	Warnings or errors occurred and no output was created.

CONSEQUENCES OF ERRORS

If an error is detected, no permanent output shall be created.

If warnings occur, permanent output shall be created if the **–c** option was specified. The following conditions shall cause warning messages to be issued:

- If a symbolic name not found in the *charmap* file is used for the descriptions of the *LC_CTYPE* or *LC_COLLATE* categories (for other categories, this shall be an error condition).

- If the number of operands to the **order** keyword exceeds the {COLL_WEIGHTS_MAX} limit.
- If optional keywords not supported by the implementation are present in the source.
- If a non-printable character has a width specified other than −1.

Other implementation-defined conditions may also cause warnings.

APPLICATION USAGE

The *charmap* definition is optional, and is contained outside the locale definition. This allows both completely self-defined source files, and generic sources (applicable to more than one codeset). To aid portability, all *charmap* definitions must use the same symbolic names for the portable character set. As explained in the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.4, Character Set Description File, it is implementation-defined whether or not users or applications can provide additional character set description files. Therefore, the −**f** option might be operable only when an implementation-defined *charmap* is named.

EXAMPLES

None.

RATIONALE

The output produced by the *localedef* utility is implementation-defined. The *name* operand is used to identify the specific locale. (As a consequence, although several categories can be processed in one execution, only categories belonging to the same locale can be processed.)

FUTURE DIRECTIONS

None.

SEE ALSO

locale, the Base Definitions volume of IEEE Std 1003.1-200x, Section 7.3, Locale Definition

CHANGE HISTORY

First released in Issue 4.

Issue 6

The −**u** option is added, as specified in the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

logger — log messages

SYNOPSIS

`logger` *`string`* `...`

DESCRIPTION

The *logger* utility saves a message, in an unspecified manner and format, containing the *string* operands provided by the user. The messages are expected to be evaluated later by personnel performing system administration tasks.

It is implementation-defined whether messages written in locales other than the POSIX locale are effective.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

string	One of the string arguments whose contents are concatenated together, in the order specified, separated by single <space>s.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *logger*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error. (This means diagnostics from *logger* to the user or application, not diagnostic messages that the user is sending to the system administrator.)
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

Unspecified.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility allows logging of information for later use by a system administrator or programmer in determining why non-interactive utilities have failed. The locations of the saved messages, their format, and retention period are all unspecified. There is no method for a conforming application to read messages, once written. |

EXAMPLES

A batch application, running non-interactively, tries to read a configuration file and fails; it may attempt to notify the system administrator with:

```
logger myname: unable to read file foo. [timestamp]
```

RATIONALE

The standard developers believed strongly that some method of alerting administrators to errors was necessary. The obvious example is a batch utility, running non-interactively, that is unable to read its configuration files or that is unable to create or write its results file. However, the standard developers did not wish to define the format or delivery mechanisms as they have historically been (and will probably continue to be) very system-specific, as well as involving functionality clearly outside of the scope of this volume of IEEE Std 1003.1-200x.

The text with *LC_MESSAGES* about diagnostic messages means diagnostics from *logger* to the user or application, not diagnostic messages that the user is sending to the system administrator.

Multiple *string* arguments are allowed, similar to *echo*, for ease-of-use.

Like the utilities *mailx* and *lp*, *logger* is admittedly difficult to test. This was not deemed sufficient justification to exclude these utilities from this volume of IEEE Std 1003.1-200x. It is also arguable that they are, in fact, testable, but that the tests themselves are not portable.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *write*

CHANGE HISTORY

First released in Issue 4.

NAME

logname — return the user's login name

SYNOPSIS

```
logname
```

DESCRIPTION

The *logname* utility shall write the user's login name to standard output. The login name shall be the string that would be returned by the *getlogin*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. Under the conditions where the *getlogin*( ) function would fail, the *logname* utility shall write a diagnostic message to standard error and exit with a non-zero exit status.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *logname*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *logname* utility output shall be a single line consisting of the user's login name:

```
"%s\n", <login name>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *logname* utility explicitly ignores the *LOGNAME* environment variable because environment changes could produce erroneous results.

EXAMPLES

None.

RATIONALE

The **passwd** file is not listed as required because the implementation may have other means of mapping login names.

FUTURE DIRECTIONS

None.

SEE ALSO

id, *who*

CHANGE HISTORY

First released in Issue 2.

NAME

lp — send files to a printer

SYNOPSIS

```
lp [−c][−d dest][−n copies][−msw][−o option]... [−t title][file...]
```

DESCRIPTION

The *lp* utility shall copy the input files to an output destination in an unspecified manner. The default output destination should be to a hardcopy device, such as a printer or microfilm recorder, that produces non-volatile, human-readable documents. If such a device is not available to the application, or if the system provides no such device, the *lp* utility shall exit with a non-zero exit status.

The actual writing to the output device may occur some time after the *lp* utility successfully exits. During the portion of the writing that corresponds to each input file, the implementation shall guarantee exclusive access to the device.

The *lp* utility shall associate a unique *request ID* with each request.

Normally, a banner page is produced to separate and identify each print job. This page may be suppressed by implementation-defined conditions, such as an operator command or one of the **−o** *option* values.

OPTIONS

The *lp* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c Exit only after further access to any of the input files is no longer required. The application can then safely delete or modify the files without affecting the output operation. Normally, files are not copied, but are linked whenever possible. If the **−c** option is not given, then the user should be careful not to remove any of the files before the request has been printed in its entirety. It should also be noted that in the absence of the **−c** option, any changes made to the named files after the request is made but before it is printed may be reflected in the printed output. On some implementations, **−c** may be on by default.

−d *dest* Specify a string that names the destination (*dest*). If *dest* is a printer, the request shall be printed only on that specific printer. If *dest* is a class of printers, the request shall be printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, and so on), requests for specific destinations need not be accepted. Destination names vary between systems.

If **−d** is not specified, and neither the *LPDEST* nor *PRINTER* environment variable is set, an unspecified destination is used. The **−d** *dest* option shall take precedence over *LPDEST*, which in turn shall take precedence over *PRINTER*. Results are undefined when *dest* contains a value that is not a valid destination name.

−m Send mail (see *mailx* (on page 2785)) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

−n *copies* Write *copies* number of copies of the files, where *copies* is a positive decimal integer. The methods for producing multiple copies and for arranging the multiple copies when multiple *file* operands are used are unspecified, except that each file shall be output as an integral whole, not interleaved with portions of other files.

−o *option*	Specify printer-dependent or class-dependent *option*s. Several such *option*s may be collected by specifying the **−o** option more than once.
−s	Suppress messages from *lp*.
−t *title*	Write *title* on the banner page of the output.
−w	Write a message on the user's terminal after the files have been printed. If the user is not logged in, then mail shall be sent instead.

OPERANDS

The following operand shall be supported:

file	A pathname of a file to be output. If no *file* operands are specified, or if a *file* operand is '−', the standard input shall be used. If a *file* operand is used, but the **−c** option is not specified, the process performing the writing to the output device may have user and group permissions that differ from that of the process invoking *lp*.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is '−'. See the INPUT FILES section. |

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *lp*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
LC_TIME	Determine the format and contents of date and time strings displayed in the *lp* banner page, if any.
LPDEST	Determine the destination. If the *LPDEST* environment variable is not set, the *PRINTER* environment variable shall be used. The **−d** *dest* option takes precedence over *LPDEST*. Results are undefined when **−d** is not specified and *LPDEST* contains a value that is not a valid destination name.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PRINTER	Determine the output device or destination. If the *LPDEST* and *PRINTER* environment variables are not set, an unspecified output device is used. The **−d** *dest* option and the *LPDEST* environment variable shall take precedence over *PRINTER*. Results are undefined when **−d** is not specified, *LPDEST* is unset, and

PRINTER contains a value that is not a valid device or destination name.

TZ Determine the timezone used to calculate date and time strings displayed in the *lp* banner page, if any. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *lp* utility shall write a *request ID* to the standard output, unless **−s** is specified. The format of the message is unspecified. The request ID can be used on systems supporting the historical *cancel* and *lpstat* utilities.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were processed successfully.
>0	No output device was available, or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *pr* and *fold* utilities can be used to achieve reasonable formatting for the implementation's default page size.

A conforming application can use one of the *file* operands only with the **−c** option or if the file is publicly readable and guaranteed to be available at the time of printing. This is because the standard gives the implementation the freedom to queue up the request for printing at some later time by a different process that might not be able to access the file.

EXAMPLES

1. To print file *file*:

```
lp −c file
```

2. To print multiple files with headers:

```
pr file1 file2 | lp
```

RATIONALE

The *lp* utility was designed to be a basic version of a utility that is already available in many historical implementations. The standard developers considered that it should be implementable simply as:

```
cat "$@" > /dev/lp
```

after appropriate processing of options, if that is how the implementation chose to do it and if exclusive access could be granted (so that two users did not write to the device simultaneously). Although in the future the standard developers may add other options to this utility, it should always be able to execute with no options or operands and send the standard input to an unspecified output device.

This volume of IEEE Std 1003.1-200x makes no representations concerning the format of the printed output, except that it must be ‘‘human-readable’’ and ‘‘non-volatile’’. Thus, writing by default to a disk or tape drive or a display terminal would not qualify. (Such destinations are not prohibited when **−d** *dest*, *LPDEST*, or *PRINTER* are used, however.)

This volume of IEEE Std 1003.1-200x is worded such that a ‘‘print job’’ consisting of multiple input files, possibly in multiple copies, is guaranteed to print so that any one file is not intermixed with another, but there is no statement that all the files or copies have to print out together.

The **−c** option may imply a spooling operation, but this is not required. The utility can be implemented to wait until the printer is ready and then wait until it is finished. Because of that, there is no attempt to define a queuing mechanism (priorities, classes of output, and so on).

On some historical systems, the request ID reported on the STDOUT can be used to later cancel or find the status of a request using utilities not defined in this volume of IEEE Std 1003.1-200x.

Although the historical System V *lp* and BSD *lpr* utilities have provided similar functionality, they used different names for the environment variable specifying the destination printer. Since the name of the utility here is *lp*, *LPDEST* (used by the System V *lp* utility) was given precedence over *PRINTER* (used by the BSD *lpr* utility). Since environments of users frequently contain one or the other environment variable, the *lp* utility is required to recognize both. If this was not done, many applications would send output to unexpected output devices when users moved from system to system.

Some have commented that *lp* has far too little functionality to make it worthwhile. Requests have proposed additional options or operands or both that added functionality. The requests included:

- Wording *requiring* the output to be ‘‘hardcopy’’
- A requirement for multiple printers
- Options for supporting various page-description languages

Given that a compliant system is not required to even have a printer, placing further restrictions upon the behavior of the printer is not useful. Since hardcopy format is so application-dependent, it is difficult, if not impossible, to select a reasonable subset of functionality that should be required on all compliant systems.

The term ‘‘unspecified’’ is used in this section in lieu of ‘‘implementation-defined’’ as most known implementations would not be able to make definitive statements in their conformance documents: the existence and usage of printers is very dependent on how the system administrator configures each individual system.

Since the default destination, device type, queuing mechanisms, and acceptable forms of input are all unspecified, usage guidelines for what a conforming application can do are as follows:

- Use the command in a pipeline, or with **−c**, so that there are no permission problems and the files can be safely deleted or modified.
- Limit output to text files of reasonable line lengths and printable characters and include no device-specific formatting information, such as a page description language. The meaning of ‘‘reasonable’’ in this context can only be answered as a quality-of-implementation issue, but it should be apparent from historical usage patterns in the industry and the locale. The *pr* and *fold* utilities can be used to achieve reasonable formatting for the default page size of the

implementation.

Alternatively, the application can arrange its installation in such a way that it requires the system administrator or operator to provide the appropriate information on *lp* options and environment variable values.

At a minimum, having this utility in this volume of IEEE Std 1003.1-200x tells the industry that conforming applications require a means to print output and provides at least a command name and *LPDEST* routing mechanism that can be used for discussions between vendors, application writers, and users. The use of ‘‘should’’ in the DESCRIPTION of *lp* clearly shows the intent of the standard developers, even if they cannot mandate that all systems (such as laptops) have printers.

This volume of IEEE Std 1003.1-200x does not specify what the ownership of the process performing the writing to the output device may be. If −**c** is not used, it is unspecified whether the process performing the writing to the output device has permission to read *file* if there are any restrictions in place on who may read *file* until after it is printed. Also, if −**c** is not used, the results of deleting *file* before it is printed are unspecified.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx

CHANGE HISTORY

First released in Issue 2.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, the requirement to associate a unique request ID, and the normal generation of a banner page is added.
- In the OPTIONS section:
  - The −**d** *dest* description is expanded, but references to *lpstat* are removed.
  - The −**m**, −**o**, −**s**, −**t**, and −**w** options are added.
- In the ENVIRONMENT VARIABLES section, *LC_TIME* may now affect the execution.
- The STDOUT section is added.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

NAME

ls — list directory contents

SYNOPSIS

XSI `ls [−CFRacdilqrtu1][−H | −L ][−fgmnopsx][file...]`

DESCRIPTION

For each operand that names a file of a type other than directory or symbolic link to a directory, *ls* shall write the name of the file as well as any requested, associated information. For each operand that names a file of type directory, *ls* shall write the names of files contained within the directory as well as any requested, associated information. If one of the **−d**, **−F**, or **−l** options are specified, and one of the **−H** or **−L** options are not specified, for each operand that names a file of type symbolic link to a directory, *ls* shall write the name of the file as well as any requested, associated information. If none of the **−d**, **−F**, or **−l** options are specified, or the **−H** or **−L** options are specified, for each operand that names a file of type symbolic link to a directory, *ls* shall write the names of files contained within the directory as well as any requested, associated information.

If no operands are specified, *ls* shall write the contents of the current directory. If more than one operand is specified, *ls* shall write non-directory operands first; it shall sort directory and non-directory operands separately according to the collating sequence in the current locale.

The *ls* utility shall detect infinite loops; that is, entering a previously visited directory that is an ancestor of the last file encountered. When it detects an infinite loop, *ls* shall write a diagnostic message to standard error and shall either recover its position in the hierarchy or terminate.

OPTIONS

The *ls* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−C	Write multi-text-column output with entries sorted down the columns, according to the collating sequence. The number of text columns and the column separator characters are unspecified, but should be adapted to the nature of the output device.
−F	Do not follow symbolic links named as operands unless the **−H** or **−L** options are specified. Write a slash (`'/'`) immediately after each pathname that is a directory, an asterisk (`'*'`) after each that is executable, a vertical bar (`'|'`) after each that is a FIFO, and an at sign (`'@'`) after each that is a symbolic link. For other file types, other symbols may be written.
−H	If a symbolic link referencing a file of type directory is specified on the command line, *ls* shall evaluate the file information and file type to be those of the file referenced by the link, and not the link itself; however, *ls* shall write the name of the link itself and not the file referenced by the link.
−L	Evaluate the file information and file type for all symbolic links (whether named on the command line or encountered in a file hierarchy) to be those of the file referenced by the link, and not the link itself; however, *ls* shall write the name of the link itself and not the file referenced by the link. When **−L** is used with **−l**, write the contents of symbolic links in the long format (see the STDOUT section).
−R	Recursively list subdirectories encountered.
−a	Write out all directory entries, including those whose names begin with a period (`'.'`). Entries beginning with a period shall not be written out unless explicitly

		referenced, the **–a** option is supplied, or an implementation-defined condition shall cause them to be written.
	–c	Use time of last modification of the file status information (see **<sys/stat.h>** in the System Interfaces volume of IEEE Std 1003.1-200x) instead of last modification of the file itself for sorting (**–t**) or writing (**–l**).
	–d	Do not follow symbolic links named as operands unless the **–H** or **–L** options are specified. Do not treat directories differently than other types of files. The use of **–d** with **–R** produces unspecified results.
XSI	**–f**	Force each argument to be interpreted as a directory and list the name found in each slot. This option shall turn off **–l**, **–t**, **–s**, and **–r**, and shall turn on **–a**; the order is the order in which entries appear in the directory.
XSI	**–g**	The same as **–l**, except that the owner shall not be written.
	–i	For each file, write the file's file serial number (see *stat*( ) in the System Interfaces volume of IEEE Std 1003.1-200x).
	–l	(The letter ell.) Do not follow symbolic links named as operands unless the **–H** or **–L** options are specified. Write out in long format (see the STDOUT section). When **–l** (ell) is specified, –1 (one) shall be assumed.
XSI	**–m**	Stream output format; list files across the page, separated by commas.
XSI	**–n**	The same as **–l**, except that the owner's UID and GID numbers shall be written, rather than the associated character strings.
XSI	**–o**	The same as **–l**, except that the group shall not be written.
XSI	**–p**	Write a slash (`'/'`) after each filename if that file is a directory.
	–q	Force each instance of non-printable filename characters and <tab>s to be written as the question-mark (`'?'`) character. Implementations may provide this option by default if the output is to a terminal device.
	–r	Reverse the order of the sort to get reverse collating sequence or oldest first.
XSI	**–s**	Indicate the total number of file system blocks consumed by each file displayed. The block size is implementation-defined.
	–t	Sort with the primary key being time modified (most recently modified first) and the secondary key being filename in the collating sequence.
	–u	Use time of last access (see **<sys/stat.h>** in the System Interfaces volume of IEEE Std 1003.1-200x) instead of last modification of the file for sorting (**–t**) or writing (**–l**).
XSI	**–x**	The same as **–C**, except that the multi-text-column output is produced with entries sorted across, rather than down, the columns.
	–1	(The numeric digit one.) Force output to be one entry per line.

Specifying more than one of the options in the following mutually exclusive pairs shall not be considered an error: **–C** and **–l** (ell), XSI **–m** and **–l** (ell), **–x** and **–l** (ell), **–C** and **–1** (one), **–H** and **–L**, **–c** and **–u**. The last option specified in each pair shall determine the output format.

OPERANDS

The following operand shall be supported:

file	A pathname of a file to be written. If the file specified is not found, a diagnostic message shall be output on standard error.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ls*:

COLUMNS	Determine the user's preferred column position width for writing multiple text-column output. If this variable contains a string representing a decimal integer, the *ls* utility shall calculate how many pathname text columns to write (see **−C**) based on the width provided. If *COLUMNS* is not set or invalid, an implementation-defined number of column positions shall be assumed, based on the implementation's knowledge of the output device. The column width chosen to write the names of files in any given directory shall be constant. Filenames shall not be truncated to fit into the multiple text-column output.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for character collation information in determining the pathname collation sequence.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments) and which characters are defined as printable (character class **print**).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_TIME	Determine the format and contents for date and time strings written by *ls*.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
TZ	Determine the timezone for date and time strings written by *ls*. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The default format shall be to list one entry per line to standard output; the exceptions are to terminals or when one of the **−C**, XSI **−m**, or **−x** options is specified. If the output is to a terminal, the format is implementation-defined.

XSI When −**m** is specified, the format used shall be:

```
"%s, %s, ...\n", <filename1>, <filename2>
```

where the largest number of filenames shall be written without exceeding the length of the line.

If the −**i** option is specified, the file's file serial number (see **<sys/stat.h>** in the System Interfaces volume of IEEE Std 1003.1-200x) shall be written in the following format before any other output for the corresponding entry:

```
%u ", <file serial number>
```

If the −**l** option is specified without −**L**, the following information shall be written:

```
"%s %u %s %s %u %s %s\n", <file mode>, <number of links>,
    <owner name>, <group name>, <number of bytes in the file>,
    <date and time>, <pathname>
```

If the file is a symbolic link, this information shall be about the link itself and the <*pathname*> field shall be of the form:

```
"%s −> %s", <pathname of link>, <contents of link>
```

If both −**l** and −**L** are specified, the following information shall be written:

```
"%s %u %s %s %u %s %s\n", <file mode>, <number of links>,
    <owner name>, <group name>, <number of bytes in the file>,
    <date and time>, <pathname of link>
```

where all fields except <*pathname of link*> shall be for the file resolved from the symbolic link.

XSI The −**g**, −**n**, and −**o** options use the same format as −**l**, but with omitted items and their associated <blank>s. See the OPTIONS section.

XSI In both the preceding −**l** forms, if <*owner name*> or <*group name*> cannot be determined, or if −**n** is given, they shall be replaced with their associated numeric values using the format `%u`.

The <*date and time*>, field shall contain the appropriate date and timestamp of when the file was last modified. In the POSIX locale, the field shall be the equivalent of the output of the following *date* command:

```
date "+%b %e %H:%M"
```

if the file has been modified in the last six months, or:

```
date "+%b %e %Y"
```

(where two <space>s are used between `%e` and `%Y`) if the file has not been modified in the last six months or if the modification date is in the future, except that, in both cases, the final <newline> produced by *date* shall not be included and the output shall be as if the *date* command were executed at the time of the last modification date of the file rather than the current time. When the *LC_TIME* locale category is not set to the POSIX locale, a different format and order of presentation of this field may be used.

If the file is a character special or block special file, the size of the file may be replaced with implementation-defined information associated with the device in question.

If the pathname was specified as a *file* operand, it shall be written as specified.

XSI The file mode written under the −**l**, −**g**, −**n**, and −**o** options shall consist of the following format:

```
"%c%s%s%s%c", <entry type>, <owner permissions>,
    <group permissions>, <other permissions>,
```

`<optional alternate access method flag>`

The <*optional alternate access method flag*> shall be a single <space> if there is no alternate or additional access control method associated with the file; otherwise, a printable character shall be used.

The <*entry type*> character shall describe the type of file, as follows:

`d`	Directory.
`b`	Block special file.
`c`	Character special file.
`l` (ell)	Symbolic link.
`p`	FIFO.
–	Regular file.

Implementations may add other characters to this list to represent other implementation-defined file types.

The next three fields shall be three characters each:

<*owner permissions*>
Permissions for the file owner class (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.4, File Access Permissions).

<*group permissions*>
Permissions for the file group class.

<*other permissions*>
Permissions for the file other class.

Each field shall have three character positions:

1. If `'r'`, the file is readable; if `'−'`, the file is not readable.
2. If `'w'`, the file is writable; if `'−'`, the file is not writable.
3. The first of the following that applies:

`S`	If in <*owner permissions*>, the file is not executable and set-user-ID mode is set. If in <*group permissions*>, the file is not executable and set-group-ID mode is set.
`s`	If in <*owner permissions*>, the file is executable and set-user-ID mode is set. If in <*group permissions*>, the file is executable and set-group-ID mode is set.
`x`	The file is executable or the directory is searchable.
–	None of the attributes of `'S'`, `'s'`, or `'x'` applies.

   Implementations may add other characters to this list for the third character position. Such additions shall, however, be written in lowercase if the file is executable or searchable, and in uppercase if it is not.

XSI If any of the **−l**, **−g**, **−n**, **−o**, or **−s** options is specified, each list of files within the directory shall be preceded by a status line indicating the number of file system blocks occupied by files in the directory in 512-byte units, rounded up to the next integral number of units, if necessary. In the POSIX locale, the format shall be:

`"total %u\n", <number of units in the directory>`

If more than one directory, or a combination of non-directory files and directories are written, either as a result of specifying multiple operands, or the −**R** option, each list of files within a directory shall be preceded by:

`"\n%s:\n", <directory name>`

If this string is the first thing to be written, the first <newline> shall not be written. This output shall precede the number of units in the directory.

XSI If the −**s** option is given, each file shall be written with the number of blocks used by the file. Along with −**C**, −**1**, −**m**, or −**x**, the number and a <space> shall precede the filename; with −**g**, −**l**, −**n**, or −**o**, they shall precede each line describing a file.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Many implementations use the equal sign (`'='`) to denote sockets bound to the file system for the −**F** option. Similarly, many historical implementations use the `'s'` character to denote sockets as the entry type characters for the −**l** option.

It is difficult for an application to use every part of the file modes field of *ls* −**l** in a portable manner. Certain file types and executable bits are not guaranteed to be exactly as shown, as implementations may have extensions. Applications can use this field to pass directly to a user printout or prompt, but actions based on its contents should generally be deferred, instead, to the *test* utility.

The output of *ls* (with the −**l** and related options) contains information that logically could be used by utilities such as *chmod* and *touch* to restore files to a known state. However, this information is presented in a format that cannot be used directly by those utilities or be easily translated into a format that can be used. A character has been added to the end of the permissions string so that applications at least have an indication that they may be working in an area they do not understand instead of assuming that they can translate the permissions string into something that can be used. Future issues or related documents may define one or more specific characters to be used based on different standard additional or alternative access control mechanisms.

As with many of the utilities that deal with filenames, the output of *ls* for multiple files or in one of the long listing formats must be used carefully on systems where filenames can contain embedded white space. Systems and system administrators should institute policies and user training to limit the use of such filenames.

The number of disk blocks occupied by the file that it reports varies depending on underlying file system type, block size units reported, and the method of calculating the number of blocks.

On some file system types, the number is the actual number of blocks occupied by the file (counting indirect blocks and ignoring holes in the file); on others it is calculated based on the file size (usually making an allowance for indirect blocks, but ignoring holes).

EXAMPLES

An example of a small directory tree being fully listed with *ls* –**laRF a** in the POSIX locale:

```
total 11
drwxr-xr-x   3 hlj      prog          64 Jul  4 12:07 ./
drwxrwxrwx   4 hlj      prog        3264 Jul  4 12:09 ../
drwxr-xr-x   2 hlj      prog          48 Jul  4 12:07 b/
-rwxr--r--   1 hlj      prog         572 Jul  4 12:07 foo*

a/b:
total 4
drwxr-xr-x   2 hlj      prog          48 Jul  4 12:07 ./
drwxr-xr-x   3 hlj      prog          64 Jul  4 12:07 ../
-rw-r--r--   1 hlj      prog         700 Jul  4 12:07 bar
```

RATIONALE

Some historical implementations of the *ls* utility show all entries in a directory except dot and dot-dot when a superuser invokes *ls* without specifying the –**a** option. When ‘‘normal’’ users invoke *ls* without specifying –**a**, they should not see information about any files with names beginning with period unless they were named as *file* operands.

Implementations are expected to traverse arbitrary depths when processing the –**R** option. The only limitation on depth should be based on running out of physical storage for keeping track of untraversed directories.

The –**1** (one) option is currently found in BSD and BSD-derived implementations only. It is required in this volume of IEEE Std 1003.1-200x so that conforming applications might ensure that output is one entry per line, even if the output is to a terminal.

Generally, this volume of IEEE Std 1003.1-200x is silent about what happens when options are given multiple times. In the cases of –**C**, –**l**, and –**1**, however, it does specify the results of these overlapping options. Since *ls* is one of the most aliased commands, it is important that the implementation perform intuitively. For example, if the alias were:

```
alias ls="ls –C"
```

and the user typed *ls* –**1**, single-text-column output should result, not an error.

The BSD *ls* provides a –**A** option (like –**a**, but dot and dot-dot are not written out). The small difference from –**a** did not seem important enough to require both.

Implementations may make –**q** the default for terminals to prevent trojan horse attacks on terminals with special escape sequences. This is not required because:

- Some control characters may be useful on some terminals; for example, a system might write them as `"\001"` or `"^A"`.
- Special behavior for terminals is not relevant to application portability.

An early proposal specified that the optional alternate access method flag had to be `'+'` if there was an alternate access method used on the file or <space> if there was not. This was changed to be <space> if there is not and a single printable character if there is. This was done for three reasons:

1. There are historical implementations using characters other than `'+'`.

2. There are implementations that vary this character used in that position to distinguish between various alternate access methods in use.
3. The standard developers did not want to preclude futures specifications that might need a way to specify more than one alternate access method.

Nonetheless, implementations providing a single alternate access method are encouraged to use '+'.

In an early proposal, the units used to specify the number of blocks occupied by files in a directory in an *ls* −**l** listing was implementation-defined. This was because BSD systems have historically used 1 024-byte units and System V systems have historically used 512-byte units. It was pointed out by BSD developers that their system has used 512-byte units in some places and 1 024-byte units in other places. (System V has consistently used 512.) Therefore, this volume of IEEE Std 1003.1-200x usually specifies 512. Future releases of BSD are expected to consistently provide 512 bytes as a default with a way of specifying 1 024-byte units where appropriate.

The <*date and time*> field in the −**l** format is specified only for the POSIX locale. As noted, the format can be different in other locales. No mechanism for defining this is present in this volume of IEEE Std 1003.1-200x, as the appropriate vehicle is a messaging system; that is, the format should be specified as a ''message''.

FUTURE DIRECTIONS

The −**s** uses implementation-defined units and cannot be used portably; it may be withdrawn in a future issue.

SEE ALSO

chmod, *find*, the System Interfaces volume of IEEE Std 1003.1-200x, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 5

Second FUTURE DIRECTION added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the −**F** option, other symbols are allowed for other file types.

Treatment of symbolic links is added, as defined in the IEEE P1003.2b draft standard.

NAME

m4 — macro processor (**DEVELOPMENT**)

SYNOPSIS

XSI
```
m4 [−s][−D name[=val]]...[−U name]... file...
```

DESCRIPTION

The *m4* utility is a macro processor that shall read one or more text files, process them according to their included macro statements, and write the results to standard output.

OPTIONS

The *m4* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the order of the **−D** and **−U** options shall be significant.

The following options shall be supported:

−s	Enable line synchronization output for the *c99* preprocessor phase (that is, **#line** directives).
−D *name*[=*val*]	Define *name* to *val* or to null if =*val* is omitted.
−U *name*	Undefine *name*.

OPERANDS

The following operand shall be supported:

file	A pathname of a text file to be processed. If no *file* is given, or if it is `'−'`, the standard input shall be read.

STDIN

The standard input shall be a text file that is used if no *file* operand is given, or if it is `'−'`.

INPUT FILES

The input file named by the *file* operand shall be a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *m4*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be the same as the input files, after being processed for macro expansion.

STDERR

The standard error shall be used to display strings with the **errprint** macro, macro tracing enabled by the **traceon** macro, the defined text for macros written by the **dumpdef** macro, or for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The *m4* utility shall compare each token from the input against the set of built-in and user-defined macros. If the token matches the name of a macro, then the token shall be replaced by the macro's defining text, if any, and rescanned for matching macro names. Once no portion of the token matches the name of a macro, it shall be written to standard output. Macros may have arguments, in which case the arguments shall be substituted into the defining text before it is rescanned.

Macro calls have the form:

```
name(arg1, arg2, ..., argn)
```

Macro names shall consist of letters, digits, and underscores, where the first character is not a digit. Tokens not of this form shall not be treated as macros.

The application shall ensure that the left parenthesis immediately follows the name of the macro. If a token matching the name of a macro is not followed by a left parenthesis, it is handled as a use of that macro without arguments.

If a macro name is followed by a left parenthesis, its arguments are the comma-separated tokens between the left parenthesis and the matching right parenthesis. Unquoted <blank>s and <newline>s preceding each argument shall be ignored. All other characters, including trailing <blank>s and <newline>s, are retained. Commas enclosed between left and right parenthesis characters do not delimit arguments.

Arguments are positionally defined and referenced. The string `"$1"` in the defining text shall be replaced by the first argument. Systems shall support at least nine arguments; only the first nine can be referenced, using the strings `"$1"` to `"$9"`, inclusive. The string `"$0"` is replaced with the name of the macro. The string `"$#"` is replaced by the number of arguments as a string. The string `"$*"` is replaced by a list of all of the arguments, separated by commas. The string `"$@"` is replaced by a list of all of the arguments separated by commas, and each argument is quoted using the current left and right quoting strings.

If fewer arguments are supplied than are in the macro definition, the omitted arguments are taken to be null. It is not an error if more arguments are supplied than are in the macro definition.

No special meaning is given to any characters enclosed between matching left and right quoting strings, but the quoting strings are themselves discarded. By default, the left quoting string consists of a grave accent (’‘’) and the right quoting string consists of an acute accent (’’’); see also the **changequote** macro.

Comments are written but not scanned for matching macro names; by default, the begin-comment string consists of the number sign character and the end-comment string consists of a

<newline>. See also the **changecom** and **dnl** macros.

The *m4* utility shall make available the following built-in macros. They can be redefined, but once this is done the original meaning is lost. Their values shall be null unless otherwise stated. In the descriptions below, the term *defining text* refers to the value of the macro: the second argument to the **define** macro, among other things. Except for the first argument to the **eval** macro, all numeric arguments to built-in macros shall be interpreted as decimal values. The string values produced as the defining text of the **decr**, **divnum**, **incr**, **index**, **len**, and **sysval** built-in macros shall be in the form of a decimal-constant as defined in the C language.

changecom	The **changecom** macro shall set the begin-comment and end-comment strings. With no arguments, the comment mechanism shall be disabled. With a single argument, that argument shall become the begin-comment string and the <newline> shall become the end-comment string. With two arguments, the first argument shall become the begin-comment string and the second argument shall become the end-comment string. Systems shall support comment strings of at least five characters.
changequote	The **changequote** macro shall set the begin-quote and end-quote strings. With no arguments, the quote strings shall be set to the default values (that is, ‘ ’). With a single argument, that argument shall become the begin-quote string and the <newline> shall become the end-quote string. With two arguments, the first argument shall become the begin-quote string and the second argument shall become the end-quote string. Systems shall support quote strings of at least five characters.
decr	The defining text of the **decr** macro shall be its first argument decremented by 1. It shall be an error to specify an argument containing any non-numeric characters.
define	The second argument shall become the defining text of the macro whose name is the first argument.
defn	The defining text of the **defn** macro shall be the quoted definition (using the current quoting strings) of its arguments.
divert	The *m4* utility maintains nine temporary buffers, numbered 1 to 9, inclusive. When the last of the input has been processed, any output that has been placed in these buffers shall be written to standard output in buffer-numerical order. The **divert** macro shall divert future output to the buffer specified by its argument. Specifying no argument or an argument of 0 shall resume the normal output process. Output diverted to a stream other than 0 to 9 shall be discarded. It shall be an error to specify an argument containing any non-numeric characters.
divnum	The defining text of the **divnum** macro shall be the number of the current output stream as a string.
dnl	The **dnl** macro shall cause *m4* to discard all input characters up to and including the next <newline>.
dumpdef	The **dumpdef** macro shall write the defined text to standard error for each of the macros specified as arguments, or, if no arguments are specified, for all macros.
errprint	The **errprint** macro shall write its arguments to standard error.
eval	The **eval** macro shall evaluate its first argument as an arithmetic expression, using 32-bit signed integer arithmetic. All of the C-language operators shall be supported, except for:

```
[]
->
++
--
(type)
unary *
sizeof
,
.
?:
unary &
```

and all assignment operators. It shall be an error to specify any of these operators. Precedence and associativity shall be as in the ISO C standard. Systems shall support octal and hexadecimal numbers as in the ISO C standard. The second argument, if specified, shall set the radix for the result; the default is 10. The third argument, if specified, sets the minimum number of digits in the result. It shall be an error to specify the second or third argument containing any non-numeric characters.

ifdef If the first argument to the **ifdef** macro is defined, the defining text shall be the second argument. Otherwise, the defining text shall be the third argument, if specified, or the null string, if not.

ifelse The **ifelse** macro takes three or more arguments. If the first two arguments compare as equal strings (after macro expansion of both arguments), the defining text shall be the third argument. If the first two arguments do not compare as equal strings and there are three arguments, the defining text shall be null. If the first two arguments do not compare as equal strings and there are four or five arguments, the defining text shall be the fourth argument. If the first two arguments do not compare as equal strings and there are six or more arguments, the first three arguments shall be discarded and processing shall restart with the remaining arguments.

include The defining text for the **include** macro shall be the contents of the file named by the first argument. It shall be an error if the file cannot be read.

incr The defining text of the **incr** macro shall be its first argument incremented by 1. It shall be an error to specify an argument containing any non-numeric characters.

index The defining text of the **index** macro shall be the first character position (as a string) in the first argument where a string matching the second argument begins (zero origin), or −1 if the second argument does not occur.

len The defining text of the **len** macro shall be the length (as a string) of the first argument.

m4exit Exit from the *m4* utility. If the first argument is specified, it is the exit code. The default is zero. It shall be an error to specify an argument containing any non-numeric characters.

m4wrap The first argument shall be processed when EOF is reached. If the **m4wrap** macro is used multiple times, the arguments specified shall be processed in the order in which the **m4wrap** macros were processed.

maketemp The defining text shall be the first argument, with any trailing `'X'` characters replaced with the current process ID as a string.

popdef	The **popdef** macro shall delete the current definition of its arguments, replacing that definition with the previous one. If there is no previous definition, the macro is undefined.
pushdef	The **pushdef** macro shall be equivalent to the **define** macro with the exception that it shall preserve any current definition for future retrieval using the **popdef** macro.
shift	The defining text for the **shift** macro shall be all of its arguments except for the first one.
sinclude	The **sinclude** macro shall be equivalent to the **include** macro, except that it shall not be an error if the file is inaccessible.
substr	The defining text for the **substr** macro shall be the substring of the first argument beginning at the zero-offset character position specified by the second argument. The third argument, if specified, shall be the number of characters to select; if not specified, the characters from the starting point to the end of the first argument shall become the defining text. It shall not be an error to specify a starting point beyond the end of the first argument and the defining text shall be null. It shall be an error to specify an argument containing any non-numeric characters.
syscmd	The **syscmd** macro shall interpret its first argument as a shell command line. The defining text shall be the string result of that command. No output redirection shall be performed by the *m4* utility. The exit status value from the command can be retrieved using the **sysval** macro.
sysval	The defining text of the **sysval** macro shall be the exit value of the utility last invoked by the **syscmd** macro (as a string).
traceon	The **traceon** macro shall enable tracing for the macros specified as arguments, or, if no arguments are specified, for all macros. The trace output shall be written to standard error in an unspecified format.
traceoff	The **traceoff** macro shall disable tracing for the macros specified as arguments, or, if no arguments are specified, for all macros.
translit	The defining text of the **translit** macro shall be the first argument with every character that occurs in the second argument replaced with the corresponding character from the third argument.
undefine	The **undefine** macro shall delete all definitions (including those preserved using the **pushdef** macro) of the macros named by its arguments.
undivert	The **undivert** macro shall cause immediate output of any text in temporary buffers named as arguments, or all temporary buffers if no arguments are specified. Buffers can be undiverted into other temporary buffers. Undiverting shall discard the contents of the temporary buffer. It shall be an error to specify an argument containing any non-numeric characters.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred

If the **m4exit** macro is used, the exit value can be specified by the input file.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The **defn** macro is useful for renaming macros, especially built-ins.

EXAMPLES

An example of a single *m4* input file capable of generating two output files follows. The file **file1.m4** could contain lines such as:

```
if(VER, 1, do_something)
if(VER, 2, do_something)
```

The makefile for the program might include:

```
file1.1.c : file1.m4
            m4 −D VER=1 file1.m4 > file1.1.c
            ...
file1.2.c : file1.m4
            m4 −D VER=2 file1.m4 > file1.2.c
            ...
```

The −**U** option can be used to undefine **VER**. If **file1.m4** contains:

```
if(VER, 1,  do_something)
if(VER, 2,  do_something)
ifndef(VER, do_something)
```

then the makefile would contain:

```
file1.0.c : file1.m4
            m4 −U VER file1.m4 > file1.0.c
            ...
file1.1.c : file1.m4
            m4 −D VER=1 file1.m4 > file1.1.c
            ...
file1.2.c : file1.m4
            m4 −D VER=2 file1.m4 > file1.2.c
            ...
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

c99

CHANGE HISTORY

First released in Issue 2.

Issue 5

The phrase “the defined text for macros written by the **dumpdef** macro” is added to the description of STDERR, and the description of **dumpdef** is updated to indicate that output is written to standard error. The description of **eval** is updated to indicate that the list of excluded C operators excludes unary '&' and '.'. In the description of **ifdef**, the phrase “and it is not defined to be zero” is deleted.

Issue 6

In the EXTENDED DESCRIPTION, the **eval** text is updated to include a `'&'` character in the excepted list.

The EXTENDED DESCRIPTION of **divert** is updated to clarify that there are only nine diversion buffers.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The Open Group Base Resolution bwg2000-006 is applied.

NAME

mailx — process messages

SYNOPSIS

Send Mode

```
mailx [−s subject] address...
```

Receive Mode

```
mailx −e
mailx [−HiNn][−F][−u user]
mailx −f[−HiNn][−F][file]
```

DESCRIPTION

The *mailx* utility provides a message sending and receiving facility. It has two major modes, selected by the options used: Send Mode and Receive Mode.

On systems that do not support the User Portability Utilities option, an application using *mailx* shall have the ability to send messages in an unspecified manner (Send Mode). Unless the first character of one or more lines is tilde (`'˜'`), all characters in the input message shall appear in the delivered message, but additional characters may be inserted in the message before it is retrieved.

On systems supporting the User Portability Utilities option, mail-receiving capabilities and other interactive features, Receive Mode, described below, also shall be enabled.

Send Mode

Send Mode can be used by applications or users to send messages from the text in standard input.

Receive Mode

Receive Mode is more oriented to interactive users. Mail can be read and sent in this interactive mode.

When reading mail, *mailx* provides commands to facilitate saving, deleting, and responding to messages. When sending mail, *mailx* allows editing, reviewing, and other modification of the message as it is entered.

Incoming mail shall be stored in one or more unspecified locations for each user, collectively called the system *mailbox* for that user. When *mailx* is invoked in Receive Mode, the system mailbox shall be the default place to find new mail. As messages are read, they shall be marked to be moved to a secondary file for storage, unless specific action is taken. This secondary file is called the **mbox** and is normally located in the directory referred to by the *HOME* environment variable (see *MBOX* in the ENVIRONMENT VARIABLES section for a description of this file). Messages shall remain in this file until explicitly removed. When the **−f** option is used to read mail messages from secondary files, messages shall be retained in those files unless specifically removed. All three of these locations—system mailbox, **mbox**, and secondary file—are referred to in this section as simply ‘‘mailboxes’’, unless more specific identification is required.

OPTIONS

The *mailx* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported. (Only the **−s** *subject* option shall be required on all systems. The other options are required only on systems supporting the User Portability Utilities option.)

−e	Test for the presence of mail in the system mailbox. The *mailx* utility shall write nothing and exit with a successful return code if there is mail to read.
−f	Read messages from the file named by the *file* operand instead of the system mailbox. (See also **folder**.) If no *file* operand is specified, read messages from the **mbox** instead of the system mailbox.
−F	Record the message in a file named after the first recipient. The name is the login-name portion of the address found first on the **To:** line in the mail header. Overrides the **record** variable, if set (see **Internal Variables in mailx** (on page 2792).)
−H	Write a header summary only.
−i	Ignore interrupts. (See also **ignore**).
−n	Do not initialize from the system default start-up file. See the EXTENDED DESCRIPTION section.
−N	Do not write an initial header summary.
−s *subject*	Set the **Subject** header field to *subject*. All characters in the *subject* string shall appear in the delivered message. The results are unspecified if *subject* is longer than {LINE_MAX} − 10 bytes or contains a <newline>.
−u *user*	Read the system mailbox of the login name *user*. This shall only be successful if the invoking user has the appropriate privileges to read the system mailbox of that user.

OPERANDS

The following operands shall be supported:

address	Addressee of message. When **−n** is specified and no user start-up files are accessed (see the EXTENDED DESCRIPTION section), the user or application shall ensure this is an address to pass to the mail delivery system. Any system or user start-up files may enable aliases (see **alias** under **Commands in mailx** (on page 2795)) that may modify the form of *address* before it is passed to the mail delivery system.
file	A pathname of a file to be read instead of the system mailbox when **−f** is specified. The meaning of the *file* option-argument shall be affected by the contents of the **folder** internal variable; see **Internal Variables in mailx** (on page 2792).

STDIN

When *mailx* is invoked in Send Mode (the first synopsis line), standard input shall be the message to be delivered to the specified addresses. When in Receive Mode, user commands shall be accepted from *stdin*. If the User Portability Utilities option is not supported, standard input lines beginning with a tilde ('~') character produce unspecified results.

If the User Portability Utilities option is supported, then in both Send and Receive Modes, standard input lines beginning with the escape character (usually tilde ('~')) shall affect processing as described in **Command Escapes in mailx** (on page 2803).

INPUT FILES

When *mailx* is used as described by this volume of IEEE Std 1003.1-200x, the *file* option-argument (see the **–f** option) and the **mbox** shall be text files containing mail messages, formatted as described in the OUTPUT FILES section. The nature of the system mailbox is unspecified; it need not be a file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *mailx*:

DEAD	Determine the pathname of the file in which to save partial messages in case of interrupts or delivery errors. The default shall be **dead.letter** in the directory named by the *HOME* variable. The behavior of *mailx* in saving partial messages is unspecified if the User Portability Utilities option is not supported and *DEAD* is not defined with the value **/dev/null**.
EDITOR	Determine the name of a utility to invoke when the **edit** (see **Commands in mailx** (on page 2795)) or **˜e** (see **Command Escapes in mailx** (on page 2803)) command is used. The default editor is unspecified. XSI On XSI-conformant systems it is *ed*. The effects of this variable are unspecified if the User Portability Utilities option is not supported.
HOME	Determine the pathname of the user's home directory.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the handling of case-insensitive address and header-field comparisons.
LC_TIME	Determine the format and contents of the date and time strings written by *mailx*.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
LISTER	Determine a string representing the command for writing the contents of the **folder** directory to standard output when the **folders** command is given (see **folders** in **Commands in mailx** (on page 2795)). Any string acceptable as a *command_string* operand to the *sh* **–c** command shall be valid. If this variable is null or not set, the output command shall be *ls*. The effects of this variable are unspecified if the User Portability Utilities option is not supported.
MAILRC	Determine the pathname of the start-up file. The default shall be **.mailrc** in the directory referred to by the *HOME* environment variable. The behavior of *mailx* is unspecified if the User Portability Utilities option is not supported and *MAILRC* is not defined with the value **/dev/null**.
MBOX	Determine a pathname of the file to save messages from the system mailbox that have been read. The **exit** command shall override this function, as shall saving the message explicitly in another file. The default shall be **mbox** in the directory

	named by the *HOME* variable. The effects of this variable are unspecified if the User Portability Utilities option is not supported.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
PAGER	Determine a string representing an output filtering or pagination command for writing the output to the terminal. Any string acceptable as a *command_string* operand to the *sh* **–c** command shall be valid. When standard output is a terminal device, the message output shall be piped through the command if the *mailx* internal variable **crt** is set to a value less the number of lines in the message; see **Internal Variables in mailx** (on page 2792). If the *PAGER* variable is null or not set, the paginator shall be either *more* or another paginator utility documented in the system documentation. The effects of this variable are unspecified if the User Portability Utilities option is not supported.
SHELL	Determine the name of a preferred command interpreter. The default shall be *sh*. The effects of this variable are unspecified if the User Portability Utilities option is not supported.
TERM	Determine the name of the terminal type, to indicate in an unspecified manner, if the internal variable **screen** is not specified, the number of lines in a screenful of headers. If *TERM* is not set or is set to null, an unspecified default terminal type shall be used and the value of a screenful is unspecified. The effects of this variable are unspecified if the User Portability Utilities option is not supported.
TZ	This variable may determine the timezone used to calculate date and time strings written by *mailx*. If *TZ* is unset or null, an unspecified default timezone shall be used.
VISUAL	Determine a pathname of a utility to invoke when the **visual** command (see **Commands in mailx** (on page 2795)) or **˜v** command-escape (see **Command Escapes in mailx** (on page 2803)) is used. If this variable is null or not set, the full-screen editor shall be *vi*. The effects of this variable are unspecified if the User Portability Utilities option is not supported.

ASYNCHRONOUS EVENTS

When *mailx* is in Send Mode and standard input is not a terminal, it shall take the standard action for all signals.

In Receive Mode, or in Send Mode when standard input is a terminal, if a SIGINT signal is received:

1. If in command mode, the current command, if there is one, shall be aborted, and a command-mode prompt shall be written.
2. If in input mode:
    a. If **ignore** is set, *mailx* shall write `"@\n"`, discard the current input line, and continue processing, bypassing the message-abort mechanism described in item 2b.
    b. If the interrupt was received while sending mail, either when in Receive Mode or in Send Mode, a message shall be written, and another subsequent interrupt, with no other intervening characters typed, shall be required to abort the mail message. If in Receive Mode and another interrupt is received, a command-mode prompt shall be written. If in Send Mode and another interrupt is received, *mailx* shall terminate with a non-zero status.

    In both cases listed in item b, if the message is not empty:

i. If **save** is enabled and the file named by *DEAD* can be created, the message shall be written to the file named by *DEAD*. If the file exists, the message shall be written to replace the contents of the file.

ii. If **save** is not enabled, or the file named by *DEAD* cannot be created, the message shall not be saved.

The *mailx* utility shall take the standard action for all other signals.

STDOUT

In command and input modes, all output, including prompts and messages, shall be written to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Various *mailx* commands and command escapes can create or add to files, including the **mbox**, the dead-letter file, and secondary mailboxes. When *mailx* is used as described in this volume of IEEE Std 1003.1-200x, these files shall be text files, formatted as follows:

```
line beginning with From<space>
[one or more header-lines; see Commands in mailx (on page 2795)]
empty line
[zero or more body lines
empty line]
[line beginning with From<space>...]
```

where each message begins with the **From** **<space>** line shown, preceded by the beginning of the file or an empty line. (The **From** **<space>** line is considered to be part of the message header, but not one of the header-lines referred to in **Commands in mailx** (on page 2795); thus, it shall not be affected by the **discard**, **ignore**, or **retain** commands.) The formats of the remainder of the **From** **<space>** line and any additional header lines are unspecified, except that none shall be empty. The format of a message body line is also unspecified, except that no line following an empty line shall start with **From** **<space>**; *mailx* shall modify any such user-entered message body lines (following an empty line and beginning with **From** **<space>**) by adding one or more characters to precede the `'F'`; it may add these characters to **From** **<space>** lines that are not preceded by an empty line.

When a message from the system mailbox or entered by the user is not a text file, it is implementation-defined how such a message is stored in files written by *mailx*.

EXTENDED DESCRIPTION

The entire EXTENDED DESCRIPTION section shall apply only to implementations supporting the User Portability Utilities option.

The *mailx* utility cannot guarantee support for all character encodings in all circumstances. For example, inter-system mail may be restricted to 7-bit data by the underlying network, 8-bit data need not be portable to non-internationalized systems, and so on. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used.

When *mailx* is invoked using one of the Receive Mode synopsis forms, it shall write a page of header-summary lines (if **−N** was not specified and there are messages, see below), followed by a prompt indicating that *mailx* can accept regular commands (see **Commands in mailx** (on page 2795)); this is termed *command mode*. The page of header-summary lines shall contain the first new message if there are new messages, or the first unread message if there are unread messages, or the first message. When *mailx* is invoked using the Send Mode synopsis and standard input is a terminal, if no subject is specified on the command line and the **asksub** variable is set, a prompt for the subject shall be written. At this point, *mailx* shall be in input mode. This input mode shall also be entered when using one of the Receive Mode synopsis forms and a reply or new message is composed using the **reply**, **Reply**, **followup**, **Followup**, or **mail** commands and standard input is a terminal. When the message is typed and the end of message is encountered, the message shall be passed to the mail delivery software. Commands can be entered by beginning a line with the escape character (by default, tilde (`'˜'`)) followed by a single command letter and optional arguments. See **Commands in mailx** (on page 2795) for a summary of these commands. It is unspecified what effect these commands will have if standard input is not a terminal when a message is entered using either the Send Mode synopsis, or the Read Mode commands **reply**, **Reply**, **followup**, **Followup**, or **mail**.

Note: For notational convenience, this section uses the default escape character, tilde, in all references and examples.

At any time, the behavior of *mailx* shall be governed by a set of environmental and internal variables. These are flags and valued parameters that can be set and cleared via the *mailx* **set** and **unset** commands.

Regular commands are of the form:

```
[command] [msglist] [argument ...]
```

If no *command* is specified in command mode, **next** shall be assumed. In input mode, commands shall be recognized by the escape character, and lines not treated as commands shall be taken as input for the message.

In command mode, each message shall be assigned a sequential number, starting with 1.

All messages have a state that shall affect how they are displayed in the header summary and how they are retained or deleted upon termination of *mailx*. There is at any time the notion of a *current* message, which shall be marked by a `'>'` at the beginning of a line in the header summary. When *mailx* is invoked using one of the Receive Mode synopsis forms, the current message shall be the first new message, if there is a new message, or the first unread message if there is an unread message, or the first message if there are any messages, or unspecified if there are no messages in the mailbox. Each command that takes an optional list of messages (*msglist*) or an optional single message (*message*) on which to operate shall leave the current message set to the highest-numbered message of the messages specified, unless the command deletes messages, in which case the current message shall be set to the first undeleted message (that is, a message not in the deleted state) after the highest-numbered message deleted by the command, if one exists, or the first undeleted message before the highest-numbered message deleted by the command, if one exists, or to an unspecified value if there are no remaining undeleted messages. All messages shall be in one of the following states:

new	The message is present in the system mailbox and has not been viewed by the user or moved to any other state. Messages in state *new* when *mailx* quits shall be retained in the system mailbox.
unread	The message has been present in the system mailbox for more than one invocation of *mailx* and has not been viewed by the user or moved to any other state. Messages in state *unread* when *mailx* quits shall be retained in the system mailbox.
read	The message has been processed by one of the following commands: **˜f**, **˜m**, **˜F**, **˜M**, **copy**, **mbox**, **next**, **pipe**, **print**, **Print**, **top**, **type**, **Type**, **undelete**. The **delete**, **dp**, and **dt** commands may also cause the next message to be marked as *read*, depending on the value of the **autoprint** variable. Messages that are in the system mailbox and in state *read* when *mailx* quits shall be saved in the **mbox**, unless the internal variable **hold** was set. Messages that are in the **mbox** or in a secondary mailbox and in state

	read when *mailx* quits shall be retained in their current location.
deleted	The message has been processed by one of the following commands: **delete**, **dp**, **dt**. Messages in state *deleted* when *mailx* quits shall be deleted. Deleted messages shall be ignored until *mailx* quits or changes mailboxes or they are specified to the undelete command; for example, the message specification /*string* shall only search the subject lines of messages that have not yet been deleted, unless the command operating on the list of messages is **undelete**. No deleted message or deleted message header shall be displayed by any *mailx* command other than **undelete**.
preserved	The message has been processed by a **preserve** command. When *mailx* quits, the message shall be retained in its current location.
saved	The message has been processed by one of the following commands: **save** or **write**. If the current mailbox is the system mailbox, and the internal variable **keepsave** is set, messages in the state saved shall be saved to the file designated by the *MBOX* variable (see the ENVIRONMENT VARIABLES section). If the current mailbox is the system mailbox, messages in the state *saved* shall be deleted from the current mailbox, when the **quit** or **file** command is used to exit the current mailbox.

The header-summary line for each message shall indicate the state of the message.

Many commands take an optional list of messages (*msglist*) on which to operate, which defaults to the current message. A *msglist* is a list of message specifications separated by <blank>s, which can include:

`n`	Message number *n*.
`+`	The next undeleted message, or the next deleted message for the **undelete** command.
–	The next previous undeleted message, or the next previous deleted message for the **undelete** command.
`.`	The current message.
`^`	The first undeleted message, or the first deleted message for the **undelete** command.
`$`	The last message.
`*`	All messages.
`n-m`	An inclusive range of message numbers.
address	All messages from *address*; any address as shown in a header summary shall be matchable in this form.
/*string*	All messages with *string* in the subject line (case ignored).
`:c`	All messages of type *c*, where *c* shall be one of:

`d`	Deleted messages.
`n`	New messages.
`o`	Old messages (any not in state *read* or *new*).
`r`	Read messages.
`u`	Unread messages.

Other commands take an optional message (*message*) on which to operate, which defaults to the current message. All of the forms allowed for *msglist* are also allowed for *message*, but if more than one message is specified, only the first shall be operated on.

Other arguments are usually arbitrary strings whose usage depends on the command involved.

Start-Up in mailx

At start-up time, *mailx* shall take the following steps in sequence:

1. Establish all variables at their stated default values.
2. Process command line options, overriding corresponding default values.
3. Import any of the *DEAD*, *EDITOR*, *MBOX*, *LISTER*, *PAGER*, *SHELL*, or *VISUAL* variables that are present in the environment, overriding the corresponding default values.
4. Read *mailx* commands from an unspecified system start-up file, unless the −**n** option is given, to initialize any internal *mailx* variables and aliases.
5. Process the start-up file of *mailx* commands named in the user *MAILRC* variable.

Most regular *mailx* commands are valid inside start-up files, the most common use being to set up initial display options and alias lists. The following commands shall be invalid in the start-up file: **!**, **edit**, **hold**, **mail**, **preserve**, **reply**, **Reply**, **shell**, **visual**, **Copy**, **followup**, and **Followup**. Any errors in the start-up file shall either cause *mailx* to terminate with a diagnostic message and a non-zero status or to continue after writing a diagnostic message, ignoring the remainder of the lines in the start-up file.

A blank line in a start-up file shall be ignored.

Internal Variables in mailx

The following variables are internal *mailx* variables. Each internal variable can be set via the *mailx* **set** command at any time. The **unset** and **set no** *name* commands can be used to erase variables.

In the following list, variables shown as:

`variable`

represent Boolean values. Variables shown as:

`variable=`*`value`*

shall be assigned string or numeric values. For string values, the rules in **Commands in mailx** (on page 2795) concerning filenames and quoting shall also apply. |

The defaults specified here may be changed by the implementation-defined system start-up file unless the user specifies the −**n** option.

allnet All network names whose login name components match shall be treated as identical. This shall cause the *msglist* message specifications to behave similarly. The default shall be **noallnet**. See also the **alternates** command and the **metoo** variable. |

append Append messages to the end of the **mbox** file upon termination instead of placing them at the beginning. The default shall be **noappend**. This variable shall not affect the **save** command when saving to the **mbox**.

ask, **asksub**
Prompt for a subject line on outgoing mail if one is not specified on the command

line with the **–s** option. The **ask** and **asksub** forms are synonyms; the system shall refer to **asksub** and **noasksub** in its messages, but shall accept **ask** and **noask** as user input to mean **asksub** and **noasksub**. It shall not be possible to set both **ask** and **noasksub**, or **noask** and **asksub**. The default shall be **asksub**, but no prompting shall be done if standard input is not a terminal.

askbcc	Prompt for the blind copy list. The default shall be **noaskbcc**.
askcc	Prompt for the copy list. The default shall be **noaskcc**.
autoprint	Enable automatic writing of messages after **delete** and **undelete** commands. The default shall be **noautoprint**.
bang	Enable the special-case treatment of exclamation marks (`'!'`) in escape command lines; see the **escape** command and **Command Escapes in mailx** (on page 2803). The default shall be **nobang**, disabling the expansion of `'!'` in the *command* argument to the **~!** command and the **~<!***command* escape.
cmd=*command*	Set the default command to be invoked by the **pipe** command. The default shall be **nocmd**.
crt=*number*	Pipe messages having more than *number* lines through the command specified by the value of the *PAGER* variable. The default shall be **nocrt**. If it is set to null, the value used is implementation-defined.
XSI **debug**	Enable verbose diagnostics for debugging. Messages are not delivered. The default shall be **nodebug**.
dot	When **dot** is set, a period on a line by itself during message input from a terminal shall also signify end-of-file (in addition to normal end-of-file). The default shall be **nodot**. If **ignoreeof** is set (see below), a setting of **nodot** shall be ignored and the period is the only method to terminate input mode.
escape=*c*	Set the command escape character to be the character `'c'`. By default, the command escape character shall be tilde. If **escape** is unset, tilde shall be used; if it is set to null, command escaping shall be disabled.
flipr	Reverse the meanings of the **R** and **r** commands. The default shall be **noflipr**.
folder=*directory*	The default directory for saving mail files. User-specified filenames beginning with a plus sign (`'+'`) shall be expanded by preceding the filename with this directory name to obtain the real pathname. If *directory* does not start with a slash (`'/'`), the contents of *HOME* shall be prefixed to it. The default shall be **nofolder**. If **folder** is unset or set to null, user-specified filenames beginning with `'+'` shall refer to files in the current directory that begin with the literal `'+'` character. See also **outfolder** below. The **folder** value need not affect the processing of the files named in *MBOX* and *DEAD*.
header	Enable writing of the header summary when entering *mailx* in Receive Mode. The default shall be **header**.
hold	Preserve all messages that are read in the system mailbox instead of putting them in the **mbox** save file. The default shall be **nohold**.
ignore	Ignore interrupts while entering messages. The default shall be **noignore**.
ignoreeof	Ignore normal end-of-file during message input. Input can be terminated only by entering a period (`'.'`) on a line by itself or by the **~.** command escape. The default

	shall be **noignoreeof**. See also **dot** above.
indentprefix=*string*	A string that shall be added as a prefix to each line that is inserted into the message by the **˜m** command escape. This variable shall default to one <tab>.
keep	When a system mailbox, secondary mailbox, or **mbox** is empty, truncate it to zero length instead of removing it. The default shall be **nokeep**.
keepsave	Keep the messages that have been saved from the system mailbox into other files in the file designated by the variable *MBOX*, instead of deleting them. The default shall be **nokeepsave**.
metoo	Suppress the deletion of the login name of the user from the recipient list when replying to a message or sending to a group. The default shall be **nometoo**.
XSI **onehop**	When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (that is, one hop away). The default shall be **noonehop**.
outfolder	Cause the files used to record outgoing messages to be located in the directory specified by the **folder** variable unless the pathname is absolute. The default shall be **nooutfolder**. See the **record** variable.
page	Insert a <form-feed> after each message sent through the pipe created by the **pipe** command. The default shall be **nopage**.
prompt=*string*	Set the command-mode prompt to *string*. If *string* is null or if **noprompt** is set, no prompting shall occur. The default shall be to prompt with the string `"? "`.
quiet	Refrain from writing the opening message and version when entering *mailx*. The default shall be **noquiet**.
record=*file*	Record all outgoing mail in the file with the pathname *file*. The default shall be **norecord**. See also **outfolder** above.
save	Enable saving of messages in the dead-letter file on interrupt or delivery error. See the variable *DEAD* for the location of the dead-letter file. The default shall be **save**.
screen=*number*	Set the number of lines in a screenful of headers for the **headers** and **z** commands. If **screen** is not specified, a value based on the terminal type identified by the *TERM* environment variable, the window size, the baud rate, or some combination of these shall be used.
sendwait	Wait for the background mailer to finish before returning. The default shall be **nosendwait**.
showto	When the sender of the message was the user who is invoking *mailx*, write the information from the **To:** line instead of the **From:** line in the header summary. The default shall be **noshowto**.
sign=*string*	Set the variable inserted into the text of a message when the **˜a** command escape is given. The default shall be **nosign**. The character sequences `'\t'` and `'\n'` shall be recognized in the variable as <tab>s and <newline>s, respectively. (See also **˜i** in **Command Escapes in mailx** (on page 2803).)

Sign=*string* Set the variable inserted into the text of a message when the **˜A** command escape is given. The default shall be **noSign**. The character sequences `'\t'` and `'\n'` shall be recognized in the variable as <tab>s and <newline>s, respectively.

toplines=*number*
Set the number of lines of the message to write with the **top** command. The default shall be 5.

Commands in mailx

The following *mailx* commands shall be provided. In the following list, header refers to lines from the message header, as shown in the OUTPUT FILES section. Header-line refers to lines within the header that begin with one or more non-white-space characters, immediately followed by a colon and white space and continuing until the next line beginning with a non-white-space character or an empty line. Header-field refers to the portion of a header line prior to the first colon in that line.

For each of the commands listed below, the command can be entered as the abbreviation (those characters in the Synopsis command word preceding the `'['`), the full command (all characters shown for the command word, omitting the `'['` and `']'`), or any truncation of the full command down to the abbreviation. For example, the **exit** command (shown as **ex[it]** in the Synopsis) can be entered as **ex**, **exi**, or **exit**.

The arguments to commands can be quoted, using the following methods:

- An argument can be enclosed between paired double-quotes (" ") or single-quotes (' '); any white space, shell word expansion, or backslash characters within the quotes shall be treated literally as part of the argument. A double-quote shall be treated literally within single-quotes and *vice versa*. These special properties of the quote marks shall occur only when they are paired at the beginning and end of the argument.
- A backslash outside of the enclosing quotes shall be discarded and the following character treated literally as part of the argument.
- An unquoted backslash at the end of a command line shall be discarded and the next line shall continue the command.

Filenames, where expected, shall be subjected to the process of shell word expansions (see Section 2.6 (on page 2238)); if more than a single pathname results and the command is expecting one file, the effects are unspecified. If the filename begins with an unquoted plus sign, it shall not be expanded, but treated as the named file (less the leading plus) in the **folder** directory. (See the **folder** variable.)

Declare Aliases

Synopsis:
```
a[lias] [alias [address...]]
g[roup] [alias [address...]]
```

Add the given addresses to the alias specified by *alias*. The names shall be substituted when *alias* is used as a recipient address specified by the user in an outgoing message (that is, other recipients addressed indirectly through the **reply** command shall not be substituted in this manner). Mail address alias substitution shall apply only when the alias string is used as a full address; for example, when **hlj** is an alias, *hlj@posix.com* does not trigger the alias substitution. If no arguments are given, write a listing of the current aliases to standard output. If only an *alias* argument is given, write a listing of the specified alias to standard output. These listings need not reflect the same order of addresses that were entered.

Declare Alternatives

Synopsis: `alt[ernates] name...`

(See also the **metoo** command.) Declare a list of alternative names for the user's login. When responding to a message, these names shall be removed from the list of recipients for the response. The comparison of names shall be in a case-insensitive manner. With no arguments, **alternates** shall write the current list of alternative names.

Change Current Directory

Synopsis:
```
cd [directory]
ch[dir] [directory]
```

Change directory. If *directory* is not specified, the contents of *HOME* shall be used.

Copy Messages

Synopsis:
```
c[opy] [file]
c[opy] [msglist] file
C[opy] [msglist]
```

Copy messages to the file named by the pathname *file* without marking the messages as saved. Otherwise, it shall be equivalent to the **save** command.

In the capitalized form, save the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise, it shall be equivalent to the **Save** command.

Delete Messages

Synopsis: `d[elete] [msglist]`

Mark messages for deletion from the mailbox. The deletions shall not occur until *mailx* quits (see the **quit** command) or changes mailboxes (see the **folder** command). If **autoprint** is set and there are messages remaining after the **delete** command, the current message shall be written as described for the **print** command (see the **print** command); otherwise, the *mailx* prompt shall be written.

Discard Header Fields

Synopsis:
```
di[scard] [header-field...]
ig[nore] [header-field...]
```

Suppress the specified header fields when writing messages. Specified *header-fields* shall be added to the list of suppressed header fields. Examples of header fields to ignore are **status** and **cc**. The fields shall be included when the message is saved. The **Print** and **Type** commands shall override this command. The comparison of header fields shall be in a case-insensitive manner. If no arguments are specified, write a list of the currently suppressed header fields to standard output; the listing need not reflect the same order of header fields that were entered.

If both **retain** and **discard** commands are given, **discard** commands shall be ignored.

Delete Messages and Display

Synopsis:

```
dp [msglist]
dt [msglist]
```

Delete the specified messages as described for the **delete** command, except that the **autoprint** variable shall have no effect, and the current message shall be written only if it was set to a message after the last message deleted by the command. Otherwise, an informational message to the effect that there are no further messages in the mailbox shall be written, followed by the *mailx* prompt.

Echo a String

Synopsis:

```
ec[ho] string ...
```

Echo the given strings, equivalent to the shell *echo* utility.

Edit Messages

Synopsis:

```
e[dit] [msglist]
```

Edit the given messages. The messages shall be placed in a temporary file and the utility named by the *EDITOR* variable is invoked to edit each file in sequence. The default *EDITOR* is unspecified.

The **edit** command does not modify the contents of those messages in the mailbox.

Exit

Synopsis:

```
ex[it]
x[it]
```

Exit from *mailx* without changing the mailbox. No messages shall be saved in the **mbox** (see also **quit**).

Change Folder

Synopsis:

```
fi[le] [file]
fold[er] [file]
```

Quit (see the **quit** command) from the current file of messages and read in the file named by the pathname *file*. If no argument is given, the name and status of the current mailbox shall be written.

Several unquoted special characters shall be recognized when used as *file* names, with the following substitutions:

`%`	The system mailbox for the invoking user.
`%user`	The system mailbox for *user*.
`#`	The previous file.
`&`	The current **mbox**.
`+file`	The named file in the **folder** directory. (See the **folder** variable.)

The default file shall be the current mailbox.

Display List of Folders

Synopsis: `folders`

Write the names of the files in the directory set by the **folder** variable. The command specified by the *LISTER* environment variable shall be used (see the ENVIRONMENT VARIABLES section).

Follow Up Specified Messages

Synopsis:
```
fo[llowup] [message]
F[ollowup] [msglist]
```

In the lowercase form, respond to a message, recording the response in a file whose name is derived from the author of the message. See also the **save** and **copy** commands and **outfolder**.

In the capitalized form, respond to the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line shall be taken from the first message and the response shall be recorded in a file whose name is derived from the author of the first message. See also the **Save** and **Copy** commands and **outfolder**.

Both forms shall override the **record** variable, if set. |

Display Header Summary for Specified Messages

Synopsis: `f[rom] [msglist]`

Write the header summary for the specified messages.

Display Header Summary

Synopsis: `h[eaders] [message]`

Write the page of headers that includes the message specified. If the *message* argument is not specified, the current message shall not change. However, if the *message* argument is specified, the current message shall become the message that appears at the top of the page of headers that includes the message specified. The **screen** variable sets the number of headers per page. See also the **z** command.

Help

Synopsis:
```
hel[p]
?
```

Write a summary of commands.

Hold Messages

Synopsis:
```
ho[ld] [msglist]
pre[serve] [msglist]
```

Mark the messages in *msglist* to be retained in the mailbox when *mailx* terminates. This shall override any commands that might previously have marked the messages to be deleted. During the current invocation of *mailx*, only the **delete**, **dp**, or **dt** commands shall remove the *preserve* marking of a message.

Execute Commands Conditionally

Synopsis:
```
i[f] s|r
mail-commands
el[se]
mail-commands
en[dif]
```

Execute commands conditionally, where **if s** executes the following *mail-commands*, up to an **else** or **endif**, if the program is in Send Mode, and **if r** shall cause the *mail-commands* to be executed only in Receive Mode.

List Available Commands

Synopsis: `l[ist]`

Write a list of all commands available. No explanation shall be given.

Mail a Message

Synopsis: `m[ail] address...`

Mail a message to the specified addresses or aliases.

Direct Messages to mbox

Synopsis: `mb[ox] [msglist]`

Arrange for the given messages to end up in the **mbox** save file when *mailx* terminates normally. See *MBOX*. See also the **exit** and **quit** commands.

Process Next Specified Message

Synopsis: `n[ext] [message]`

If the current message has not been written (for example, by the **print** command) since *mailx* started or since any other message was the current message, behave as if the **print** command was entered. Otherwise, if there is an undeleted message after the current message, make it the current message and behave as if the **print** command was entered. Otherwise, an informational message to the effect that there are no further messages in the mailbox shall be written, followed by the *mailx* prompt.

Pipe Message

Synopsis:
```
pi[pe] [[msglist] command]
| [[msglist] command]
```

Pipe the messages through the given *command* by invoking the command interpreter specified by *SHELL* with two arguments: −**c** and *command*. (See also *sh* −**c**.) The application shall ensure that the command is given as a single argument. Quoting, described previously, can be used to accomplish this. If no arguments are given, the current message shall be piped through the command specified by the value of the **cmd** variable. If the **page** variable is set, a <form-feed> shall be inserted after each message.

Display Message with Headers

Synopsis: `P[rint] [msglist]`
`T[ype] [msglist]`

Write the specified messages, including all header lines, to standard output. Override suppression of lines by the **discard**, **ignore**, and **retain** commands. If **crt** is set, the messages longer than the number of lines specified by the **crt** variable shall be paged through the command specified by the *PAGER* environment variable.

Display Message

Synopsis: `p[rint] [msglist]`
`t[ype] [msglist]`

Write the specified messages to standard output. If **crt** is set, the messages longer than the number of lines specified by the **crt** variable shall be paged through the command specified by the *PAGER* environment variable.

Quit

Synopsis: `q[uit]`
`end-of-file`

Terminate *mailx*, storing messages that were read in **mbox** (if the current mailbox is the system mailbox and unless **hold** is set), deleting messages that have been explicitly saved (unless **keepsave** is set), discarding messages that have been deleted, and saving all remaining messages in the mailbox.

Reply to a Message List

Synopsis: `R[eply] [msglist]`
`R[espond] [msglist]`

Mail a reply message to the sender of each message in the *msglist*. The subject line shall be formed by concatenating **Re:**<space> (unless it already begins with that string) and the subject from the first message. If **record** is set to a filename, the response shall be saved at the end of that file.

See also the **flipr** variable.

Reply to a Message

Synopsis: `r[eply] [message]`
`r[espond] [message]`

Mail a reply message to all recipients included in the header of the message. The subject line shall be formed by concatenating **Re:**<space> (unless it already begins with that string) and the subject from the message. If **record** is set to a filename, the response shall be saved at the end of that file.

See also the **flipr** variable.

Retain Header Fields

Synopsis: `ret[ain] [header-field...]`

Retain the specified header fields when writing messages. This command shall override all **discard** and **ignore** commands. The comparison of header fields shall be in a case-insensitive manner. If no arguments are specified, write a list of the currently retained header fields to standard output; the listing need not reflect the same order of header fields that were entered.

Save Messages

Synopsis:
```
s[ave] [file]
s[ave] [msglist] file
S[ave] [msglist]
```

Save the specified messages in the file named by the pathname *file*, or the **mbox** if the *file* argument is omitted. The file shall be created if it does not exist; otherwise, the messages shall be appended to the file. The message shall be put in the state *saved*, and shall behave as specified in the description of the *saved* state when the current mailbox is exited by the **quit** or **file** command.

In the capitalized form, save the specified messages in a file whose name is derived from the author of the first message. The name of the file shall be taken to be the author's name with all network addressing stripped off. See also the **Copy**, **followup**, and **Followup** commands and **outfolder** variable.

Set Variables

Synopsis: `se[t] [name[=[string]] ...] [name=number ...] [noname ...]`

Define one or more variables called *name*. The variable can be given a null, string, or numeric value. Quoting and backslash escapes can occur anywhere in *string*, as described previously, as if the *string* portion of the argument were the entire argument. The forms *name* and *name*= shall be equivalent to *name*="" for variables that take string values. The **set** command without arguments shall write a list of all defined variables and their values. The **no** *name* form shall be equivalent to **unset** *name*.

Invoke a Shell

Synopsis: `sh[ell]`

Invoke an interactive command interpreter (see also *SHELL*).

Display Message Size

Synopsis: `si[ze] [msglist]`

Write the size in bytes of each of the specified messages.

Read mailx Commands From a File

Synopsis: `so[urce] file`

Read and execute commands from the file named by the pathname *file* and return to command mode.

Display Beginning of Messages

Synopsis: `to[p] [msglist]`

Write the top few lines of each of the specified messages. If the **toplines** variable is set, it is taken as the number of lines to write. The default shall be 5.

Touch Messages

Synopsis: `tou[ch] [msglist]`

Touch the specified messages. If any message in *msglist* is not specifically deleted nor saved in a file, it shall be placed in the **mbox** upon normal termination. See **exit** and **quit**.

Delete Aliases

Synopsis: `una[lias] [alias]...`

Delete the specified alias names. If a specified alias does not exist, the results are unspecified.

Undelete Messages

Synopsis: `u[ndelete] [msglist]`

Change the state of the specified messages from deleted to read. If **autoprint** is set, the last message of those restored shall be written. If *msglist* is not specified, the message shall be selected as follows:

- If there are any deleted messages that follow the current message, the first of these shall be chosen.
- Otherwise, the last deleted message that also precedes the current message shall be chosen.

Unset Variables

Synopsis: `uns[et] name...`

Cause the specified variables to be erased.

Edit Message with Full-Screen Editor

Synopsis: `v[isual] [msglist]`

Edit the given messages with a screen editor. Each message shall be placed in a temporary file, and the utility named by the *VISUAL* variable shall be invoked to edit each file in sequence. The default editor shall be *vi*.

The **visual** command does not modify the contents of those messages in the mailbox.

Write Messages to a File

Synopsis: `w[rite] [msglist] file`

Write the given messages to the file specified by the pathname *file*, minus the message header. Otherwise, it shall be equivalent to the **save** command.

Scroll Header Display

Synopsis: `z[+|−]`

Scroll the header display forward (if `'+'` is specified or if no option is specified) or backward (if `'−'` is specified) one screenful. The number of headers written shall be set by the **screen** variable.

Invoke Shell Command

Synopsis: `!command`

Invoke the command interpreter specified by *SHELL* with two arguments: **−c** and *command*. (See also *sh* **−c**.) If the **bang** variable is set, each unescaped occurrence of `'!'` in command shall be replaced with the command executed by the previous **!** command or **˜!** command escape.

Null Command

Synopsis: `# comment`

This null command (comment) shall be ignored by *mailx*.

Display Current Message Number

Synopsis: `=`

Write the current message number.

Command Escapes in mailx

The following commands can be entered only from input mode, by beginning a line with the escape character (by default, tilde (`'˜'`)). See the **escape** variable description for changing this special character. The format for the commands shall be:

`<escape-character><command-char><separator>[<arguments>]` |

where the <*separator*> can be zero or more <blank>s. |

In the following descriptions, the application shall ensure that the argument *command* (but not *mailx-command)* is a shell command string. Any string acceptable to the command interpreter specified by the *SHELL* variable when it is invoked as *SHELL* **−c** *command_string* shall be valid. The command can be presented as multiple arguments (that is, quoting is not required).

Command escapes that are listed with *msglist* or *mailx-command* arguments are invalid in Send Mode and produce unspecified results.

~! *command* Invoke the command interpreter specified by *SHELL* with two arguments: **−c** and *command*; and then return to input mode. If the **bang** variable is set, each unescaped occurrence of `'!'` in *command* shall be replaced with the command executed by the previous **!** command or **˜!** command escape.

~. Simulate end-of-file (terminate message input).

~: *mailx-command*, **˜_** *mailx-command*
Perform the command-level request.

~? Write a summary of command escapes.

~A This shall be equivalent to **˜i Sign**.

~a This shall be equivalent to **˜i sign**.

~b *name*. . .	Add the *name*s to the blind carbon copy (**Bcc**) list.
~c *name*. . .	Add the *name*s to the carbon copy (**Cc**) list.
~d	Read in the dead-letter file. See *DEAD* for a description of this file.
~e	Invoke the editor, as specified by the *EDITOR* environment variable, on the partial message.
~f [*msglist*]	Forward the specified messages. The specified messages shall be inserted into the current message without alteration. This command escape also shall insert message headers into the message with field selection affected by the **discard**, **ignore**, and **retain** commands.
~F [*msglist*]	This shall be the equivalent of the **˜f** command escape, except that all headers shall be included in the message, regardless of previous **discard**, **ignore**, and **retain** commands.
~h	If standard input is a terminal, prompt for a **Subject** line and the **To**, **Cc**, and **Bcc** lists. Other implementation-defined headers may also be presented for editing. If the field is written with an initial value, it can be edited as if it had just been typed.
~i *string*	Insert the value of the named variable, followed by a <newline>, into the text of the message. If the string is unset or null, the message shall not be changed.
~m [*msglist*]	Insert the specified messages into the message, prefixing non-empty lines with the string in the **indentprefix** variable. This command escape also shall insert message headers into the message, with field selection affected by the **discard**, **ignore**, and **retain** commands.
~M [*msglist*]	This shall be the equivalent of the **˜m** command escape, except that all headers shall be included in the message, regardless of previous **discard**, **ignore**, and **retain** commands.
~p	Write the message being entered. If the message is longer than **crt** lines (see **Internal Variables in mailx** (on page 2792)), the output shall be paginated as described for the *PAGER* variable.
~q	Quit (see the **quit** command) from input mode by simulating an interrupt. If the body of the message is not empty, the partial message shall be saved in the dead-letter file. See *DEAD* for a description of this file.
"**˜r** *file*, **˜<** *file*, **˜r** !*command*, **˜<** !*command*"	Read in the file specified by the pathname *file*. If the argument begins with an exclamation mark (`'!'`), the rest of the string shall be taken as an arbitrary system command; the command interpreter specified by *SHELL* shall be invoked with two arguments: **−c** and *command*. The standard output of *command* shall be inserted into the message.
~s *string*	Set the subject line to *string*.
~t *name*. . .	Add the given *name*s to the **To** list.
~v	Invoke the full-screen editor, as specified by the *VISUAL* environment variable, on the partial message.
~w *file*	Write the partial message, without the header, onto the file named by the pathname *file*. The file shall be created or the message shall be appended to it if the file exists.

~x	Exit as with **~q**, except the message shall not be saved in the dead-letter file.
~| *command*	Pipe the body of the message through the given *command* by invoking the command interpreter specified by *SHELL* with two arguments: **–c** and *command*. If the *command* returns a successful exit status, the standard output of the command shall replace the message. Otherwise, the message shall remain unchanged. If the *command* fails, an error message giving the exit status shall be written.

EXIT STATUS

When the **–e** option is specified, the following exit values are returned:

0	Mail was found.
>0	Mail was not found or an error occurred.

Otherwise, the following exit values are returned:

0	Successful completion; note that this status implies that all messages were *sent*, but it gives no assurances that any of them were actually *delivered*.
>0	An error occurred.

CONSEQUENCES OF ERRORS

When in input mode (Receive Mode) or Send Mode:

- If an error is encountered processing a command escape (see **Command Escapes in mailx** (on page 2803)), a diagnostic message shall be written to standard error, and the message being composed may be modified, but this condition shall not prevent the message from being sent.
- Other errors shall prevent the sending of the message.

When in command mode:

- Default.

APPLICATION USAGE

Delivery of messages to remote systems requires the existence of communication paths to such systems. These need not exist.

Input lines are limited to {LINE_MAX} bytes, but mailers between systems may impose more severe line-length restrictions. This volume of IEEE Std 1003.1-200x does not place any restrictions on the length of messages handled by *mailx*, and for delivery of local messages the only limitations should be the normal problems of available disk space for the target mail file. When sending messages to external machines, applications are advised to limit messages to less than 100,000 bytes because some mail gateways impose message-length restrictions.

The format of the system mailbox is intentionally unspecified. Not all systems implement system mailboxes as flat files, particularly with the advent of multimedia mail messages. Some system mailboxes may be multiple files, others records in a database. The internal format of the messages themselves are specified with the historical format from Version 7, but only after they have been saved in some file other than the system mailbox. This was done so that many historical applications expecting text-file mailboxes are not broken.

Some new formats for messages can be expected in the future, probably including binary data, bit maps, and various multimedia objects. As described here, *mailx* is not prohibited from handling such messages, but it must store them as text files in secondary mailboxes (unless some extension, such as a variable or command line option, is used to change the stored format). Its method of doing so is implementation-defined and might include translating the data into text file-compatible or readable form or omitting certain portions of the message from the stored output.

The **discard** and **ignore** commands are not inverses of the **retain** command. The **retain** command discards all header-fields except those explicitly retained. The **discard** command keeps all header-fields except those explicitly discarded. If headers exist on the retained header list, **discard** and **ignore** commands are ignored.

EXAMPLES

None.

RATIONALE

The standard developers felt strongly that a method for applications to send messages to specific users was necessary. The obvious example is a batch utility, running non-interactively, that wishes to communicate errors or results to a user. However, the actual format, delivery mechanism, and method of reading the message are clearly beyond the scope of this volume of IEEE Std 1003.1-200x.

The intent of this command is to provide a simple, portable interface for sending messages non-interactively. It merely defines a ''front-end'' to the historical mail system. It is suggested that implementations explicitly denote the sender and recipient in the body of the delivered message. Further specification of formats for either the message envelope or the message itself were deliberately not made, as the industry is in the midst of changing from the current standards to a more internationalized standard and it is probably incorrect, at this time, to require either one.

Implementations are encouraged to conform to the various delivery mechanisms described in the CCITT X.400 standards or to the equivalent Internet standards, described in Internet Request for Comment (RFC) documents RFC 819, RFC 822, RFC 920, RFC 921, and RFC 1123.

Many historical systems modified each body line that started with **From** by prefixing the `'F'` with `'>'`. It is unnecessary, but allowed, to do that when the string does not follow a blank line because it cannot be confused with the next header.

The *edit* and *visual* commands merely edit the specified messages in a temporary file. They do not modify the contents of those messages in the mailbox; such a capability could be added as an extension, such as by using different command names.

The restriction on a subject line being {LINE_MAX}−10 bytes is based on the historical format that consumes 10 bytes for **Subject:** and the trailing <newline>. Many historical mailers that a message may encounter on other systems are not able to handle lines that long, however.

Like the utilities *logger* and *lp*, *mailx* admittedly is difficult to test. This was not deemed sufficient justification to exclude this utility from this volume of IEEE Std 1003.1-200x. It is also arguable that it is, in fact, testable, but that the tests themselves are not portable.

When *mailx* is being used by an application that wishes to receive the results as if none of the User Portability Utilities option features were supported, the *DEAD* environment variable must be set to **/dev/null**. Otherwise, it may be subject to the file creations described in *mailx* ASYNCHRONOUS EVENTS. Similarly, if the *MAILRC* environment variable is not set to **/dev/null**, historical versions of *mailx* and *Mail* read initialization commands from a file before processing begins. Since the initialization that a user specifies could alter the contents of messages an application is trying to send, such applications must set *MAILRC* to **/dev/null**.

The description of *LC_TIME* uses ''may affect'' because many historical implementations do not or cannot manipulate the date and time strings in the incoming mail headers. Some headers found in incoming mail do not have enough information to determine the timezone in which the mail originated, and, therefore, *mailx* cannot convert the date and time strings into the internal form that then is parsed by routines like *strftime*( ) that can take *LC_TIME* settings into account.

Changing all these times to a user-specified format is allowed, but not required.

The paginator selected when *PAGER* is null or unset is partially unspecified to allow the System V historical practice of using *pg* as the default. Bypassing the pagination function, such as by declaring that *cat* is the paginator, would not meet with the intended meaning of this description. However, any ‘‘portable user’’ would have to set *PAGER* explicitly to get his or her preferred paginator on all systems. The paginator choice was made partially unspecified, unlike the *VISUAL* editor choice (mandated to be *vi*) because most historical pagers follow a common theme of user input, whereas editors differ dramatically.

Options to specify addresses as **cc** (carbon copy) or **bcc** (blind carbon copy) were considered to be format details and were omitted.

A zero exit status implies that all messages were *sent*, but it gives no assurances that any of them were actually *delivered*. The reliability of the delivery mechanism is unspecified and is an appropriate marketing distinction between systems.

In order to conform to the Utility Syntax Guidelines, a solution was required to the optional *file* option-argument to **−f**. By making *file* an operand, the guidelines are satisfied and users remain portable. However, it does force implementations to support usage such as:

```
mailx −fin mymail.box
```

The **no** *name* method of unsetting variables is not present in all historical systems, but it is in System V and provides a logical set of commands corresponding to the format of the display of options from the *mailx set* command without arguments.

The **ask** and **asksub** variables are the names selected by BSD and System V, respectively, for the same feature. They are synonyms in this volume of IEEE Std 1003.1-200x.

The *mailx echo* command was not documented in the BSD version and has been omitted here because it is not obviously useful for interactive users.

The default prompt on the System V *mailx* is a question mark, on BSD *Mail* an ampersand. Since this volume of IEEE Std 1003.1-200x chose the *mailx* name, it kept the System V default, assuming that BSD users would not have difficulty with this minor incompatibility (that they can override).

The meanings of **r** and **R** are reversed between System V *mailx* and SunOS *Mail*. Once again, since this volume of IEEE Std 1003.1-200x chose the *mailx* name, it kept the System V default, but allows the SunOS user to achieve the desired results using **flipr**, an internal variable in System V *mailx*, although it has not been documented in the SVID

The **indentprefix** variable, the **retain** and **unalias** commands, and the **~F** and **~M** command escapes were adopted from 4.3 BSD *Mail*.

The **version** command was not included because no sufficiently general specification of the version information could be devised that would still be useful to a portable user. This command name should be used by suppliers who wish to provide version information about the *mailx* command.

The ‘‘implementation-specific (unspecified) system start-up file’’ historically has been named **/etc/mailx.rc**, but this specific name and location are not required.

The intent of the wording for the **next** command is that if any command has already displayed the current message it should display a following message, but, otherwise, it should display the current message. Consider the command sequence:

```
next 3
delete 3
```

```
next
```

where the **autoprint** option was not set. The normative text specifies that the second **next** command should display a message following the third message, because even though the current message has not been displayed since it was set by the **delete** command, it has been displayed since the current message was anything other than message number 3. This does not always match historical practice in some implementations, where the command file address followed by **next** (or the default command) would skip the message for which the user had searched.

FUTURE DIRECTIONS

None.

SEE ALSO

ed, *ls*, *more*, *vi*

CHANGE HISTORY

First released in Issue 2.

Issue 5

The description of the EDITOR environment variable is changed to indicate that *ed* is the default editor if this variable is not set. In previous issues, this default was not stated explicitly at this point but was implied further down in the text.

FUTURE DIRECTIONS section added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **–F** option is added.
- The **allnet**, **debug**, and **sendwait** internal variables are added.
- The **C**, **ec**, **fo**, **F**, and **S** *mailx* commands are added.

In the DESCRIPTION and ENVIRONMENT VARIABLES sections, text stating ‘‘*HOME* directory’’ is replaced by ‘‘directory referred to by the *HOME* environment variable’’.

The *mailx* utility is aligned with the IEEE P1003.2b draft standard, which included various clarifications to resolve IEEE PASC Interpretations submitted for the ISO POSIX-2:1993 standard. In particular, the changes here address IEEE PASC Interpretations 1003.2 #10, #11, #103, #106, #108, #114, #115, #122, and #129.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

NAME

make — maintain, update, and regenerate groups of programs (**DEVELOPMENT**)

SYNOPSIS

SD
```
make [−einpqrst][−f makefile]...[ −k| −S][macro=value]...
    [target_name...]
```

DESCRIPTION

The *make* utility shall update files that are derived from other files. A typical case is one where object files are derived from the corresponding source files. The *make* utility examines time relationships and shall update those derived files (called targets) that have modified times earlier than the modified times of the files (called prerequisites) from which they are derived. A description file (makefile) contains a description of the relationships between files, and the commands that need to be executed to update the targets to reflect changes in their prerequisites. Each specification, or rule, shall consist of a target, optional prerequisites, and optional commands to be executed when a prerequisite is newer than the target. There are two types of rule:

1. *Inference rules*, which have one target name with at least one period (`'.'`) and no slash (`'/'`)
2. *Target rules*, which can have more than one target name

In addition, *make* shall have a collection of built-in macros and inference rules that infer prerequisite relationships to simplify maintenance of programs.

To receive exactly the behavior described in this section, the user shall ensure that a portable makefile shall:

- Include the special target **.POSIX**
- Omit any special target reserved for implementations (a leading period followed by uppercase letters) that has not been specified by this section

The behavior of *make* is unspecified if either or both of these conditions are not met.

OPTIONS

The *make* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−e	Cause environment variables, including those with null values, to override macro assignments within makefiles.
−f *makefile*	Specify a different makefile. The argument *makefile* is a pathname of a description file, which is also referred to as the *makefile*. A pathname of `'−'` shall denote the standard input. There can be multiple instances of this option, and they shall be processed in the order specified. The effect of specifying the same option-argument more than once is unspecified.
−i	Ignore error codes returned by invoked commands. This mode is the same as if the special target **.IGNORE** were specified without prerequisites.
−k	Continue to update other targets that do not depend on the current target if a non-ignored error occurs while executing the commands to bring a target up-to-date.
−n	Write commands that would be executed on standard output, but do not execute them. However, lines with a plus sign (`'+'`) prefix shall be executed. In this mode,

lines with an at sign (`'@'`) character prefix shall be written to standard output.

−p Write to standard output the complete set of macro definitions and target descriptions. The output format is unspecified.

−q Return a zero exit value if the target file is up-to-date; otherwise, return an exit value of 1. Targets shall not be updated if this option is specified. However, a makefile command line (associated with the targets) with a plus sign (`'+'`) prefix shall be executed.

−r Clear the suffix list and does not use the built-in rules.

−S Terminate *make* if an error occurs while executing the commands to bring a target up-to-date. This shall be the default and the opposite of **−k**.

−s Do not write makefile command lines or touch messages (see **−t**) to standard output before executing. This mode shall be the same as if the special target **.SILENT** were specified without prerequisites.

−t Update the modification time of each target as though a *touch target* had been executed. Targets that have prerequisites but no commands (see **Target Rules** (on page 2813)), or that are already up-to-date, shall not be touched in this manner. Write messages to standard output for each target file indicating the name of the file and that it was touched. Normally, the makefile command lines associated with each target are not executed. However, a command line with a plus sign (`'+'`) prefix shall be executed.

Any options specified in the *MAKEFLAGS* environment variable shall be evaluated before any options specified on the *make* utility command line. If the **−k** and **−S** options are both specified on the *make* utility command line or by the *MAKEFLAGS* environment variable, the last option specified shall take precedence. If the **−f** or **−p** options appear in the *MAKEFLAGS* environment variable, the result is undefined.

OPERANDS

The following operands shall be supported:

target_name Target names, as defined in the EXTENDED DESCRIPTION section. If no target is specified, while *make* is processing the makefiles, the first target that *make* encounters that is not a special target or an inference rule shall be used.

macro=*value* Macro definitions, as defined in **Macros** (on page 2815).

If the *target_name* and *macro*=*value* operands are intermixed on the *make* utility command line, the results are unspecified.

STDIN

The standard input shall be used only if the *makefile* option-argument is `'−'`. See the INPUT FILES section.

INPUT FILES

The input file, otherwise known as the makefile, is a text file containing rules, macro definitions, and comments. See the EXTENDED DESCRIPTION section. |

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *make*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

MAKEFLAGS
This variable shall be interpreted as a character string representing a series of option characters to be used as the default options. The implementation shall accept both of the following formats (but need not accept them when intermixed):

- The characters are option letters without the leading hyphens or <blank> separation used on a *make* utility command line.
- The characters are formatted in a manner similar to a portion of the *make* utility command line: options are preceded by hyphens and <blank>-separated as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines. The *macro*=*value* macro definition operands can also be included. The difference between the contents of *MAKEFLAGS* and the *make* utility command line is that the contents of the variable shall not be subjected to the word expansions (see Section 2.6 (on page 2238)) associated with parsing the command line values.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

XSI *PROJECTDIR*
Provide a directory to be used to search for SCCS files not found in the current directory. In all of the following cases, the search for SCCS files is made in the directory **SCCS** in the identified directory. If the value of *PROJECTDIR* begins with a slash, it shall be considered an absolute pathname; otherwise, the value of *PROJECTDIR* is treated as a user name and that user's initial working directory shall be examined for a subdirectory **src** or **source**. If such a directory is found, it shall be used. Otherwise, the value is used as a relative pathname.

If *PROJECTDIR* is not set or has a null value, the search for SCCS files shall be made in the directory **SCCS** in the current directory.

The setting of *PROJECTDIR* affects all files listed in the remainder of this utility description for files with a component named **SCCS**.

The value of the *SHELL* environment variable shall not be used as a macro and shall not be modified by defining the **SHELL** macro in a makefile or on the command line. All other environment variables, including those with null values, shall be used as macros, as defined in **Macros** (on page 2815).

ASYNCHRONOUS EVENTS

If not already ignored, *make* shall trap SIGHUP, SIGTERM, SIGINT, and SIGQUIT and remove the current target unless the target is a directory or the target is a prerequisite of the special target **.PRECIOUS** or unless one of the **−n**, **−p**, or **−q** options was specified. Any targets removed in this manner shall be reported in diagnostic messages of unspecified format, written to standard error. After this cleanup process, if any, *make* shall take the standard action for all other signals.

STDOUT

The *make* utility shall write all commands to be executed to standard output unless the –**s** option was specified, the command is prefixed with an at sign, or the special target **.SILENT** has either the current target as a prerequisite or has no prerequisites. If *make* is invoked without any work needing to be done, it shall write a message to standard output indicating that no action was taken. If the –**t** option is present and a file is touched, *make* shall write to standard output a message of unspecified format indicating that the file was touched, including the filename of the file.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Files can be created when the –**t** option is present. Additional files can also be created by the utilities invoked by *make*.

EXTENDED DESCRIPTION

The *make* utility attempts to perform the actions required to ensure that the specified targets are up-to-date. A target is considered out-of-date if it is older than any of its prerequisites or if it does not exist. The *make* utility shall treat all prerequisites as targets themselves and recursively ensure that they are up-to-date, processing them in the order in which they appear in the rule. The *make* utility shall use the modification times of files to determine whether the corresponding targets are out-of-date.

After *make* has ensured that all of the prerequisites of a target are up-to-date and if the target is out-of-date, the commands associated with the target entry shall be executed. If there are no commands listed for the target, the target shall be treated as up-to-date.

Makefile Syntax

A makefile can contain rules, macro definitions (see **Macros** (on page 2815)), and comments. There are two kinds of rules: *inference rules* and *target rules*. The *make* utility shall contain a set of built-in inference rules. If the –**r** option is present, the built-in rules shall not be used and the suffix list shall be cleared. Additional rules of both types can be specified in a makefile. If a rule is defined more than once, the value of the rule shall be that of the last one specified. Macros can also be defined more than once, and the value of the macro is specified in **Macros** (on page 2815). Comments start with a number sign ('#') and continue until an unescaped <newline> is reached.

By default, the following files shall be tried in sequence: **./makefile** and **./Makefile**. If neither **./makefile** or **./Makefile** are found, other implementation-defined files may also be tried. XSI On XSI-conformant systems, the additional files **./s.makefile**, **SCCS/s.makefile**, **./s.Makefile**, and **SCCS/s.Makefile** shall also be tried.

The –**f** option shall direct *make* to ignore any of these default files and use the specified argument as a makefile instead. If the '–' argument is specified, standard input shall be used.

The term *makefile* is used to refer to any rules provided by the user, whether in **./makefile** or its variants, or specified by the –**f** option.

The rules in makefiles shall consist of the following types of lines: target rules, including special targets (see **Target Rules** (on page 2813)), inference rules (see **Inference Rules** (on page 2816)), macro definitions (see **Macros** (on page 2815)), empty lines, and comments.

When an escaped <newline> (one preceded by a backslash) is found anywhere in the makefile except in a command line, it shall be replaced, along with any leading white space on the following line, with a single <space>. When an escaped <newline> is found in a command line in a makefile, the command line shall contain the backslash, the <newline>, and the next line, except that the first character of the next line shall not be included if it is a <tab>.

Makefile Execution

Makefile command lines shall be processed one at a time by writing the makefile command line to the standard output (unless one of the conditions listed under `'@'` suppresses the writing) and executing the command(s) in the line. A <tab> may precede the command to standard output. Command execution shall be as if the makefile command line were the argument to the *system*( ) function. The environment for the command being executed shall contain all of the variables in the environment of *make*.

By default, when *make* receives a non-zero status from the execution of a command, it shall terminate with an error message to standard error.

Makefile command lines can have one or more of the following prefixes: a hyphen (`'−'`), an at sign (`'@'`), or a plus sign (`'+'`). These shall modify the way in which *make* processes the command. When a command is written to standard output, the prefix shall not be included in the output.

– If the command prefix contains a hyphen, or the −**i** option is present, or the special target **.IGNORE** has either the current target as a prerequisite or has no prerequisites, any error found while executing the command shall be ignored.

@ If the command prefix contains an at sign and the *make* utility command line −**n** option is not specified, or the −**s** option is present, or the special target **.SILENT** has either the current target as a prerequisite or has no prerequisites, the command shall not be written to standard output before it is executed.

\+ If the command prefix contains a plus sign, this indicates a makefile command line that shall be executed even if −**n**, −**q**, or −**t** is specified.

Target Rules

Target rules are formatted as follows:

```
target [target...]: [prerequisite...][;command]
[<tab>command
<tab>command
...]
```

```
line that does not begin with <tab>
```

Target entries are specified by a <blank>-separated, non-null list of targets, then a colon, then a <blank>-separated, possibly empty list of prerequisites. Text following a semicolon, if any, and all following lines that begin with a <tab>, are makefile command lines to be executed to update the target. The first non-empty line that does not begin with a <tab> or `'#'` shall begin a new entry. An empty or blank line, or a line beginning with `'#'`, may begin a new entry.

Applications shall select target names from the set of characters consisting solely of periods, underscores, digits, and alphabetics from the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set). Implementations may allow other characters in target names as extensions. The interpretation of targets containing the characters `'%'` and `'"'` is implementation-defined.

A target that has prerequisites, but does not have any commands, can be used to add to the prerequisite list for that target. Only one target rule for any given target can contain commands.

Lines that begin with one of the following are called *special targets* and control the operation of *make*:

.DEFAULT If the makefile uses this special target, the application shall ensure that it is specified with commands, but without prerequisites. The commands shall be used by *make* if there are no other rules available to build a target.

.IGNORE Prerequisites of this special target are targets themselves; this shall cause errors from commands associated with them to be ignored in the same manner as specified by the −**i** option. Subsequent occurrences of **.IGNORE** shall add to the list of targets ignoring command errors. If no prerequisites are specified, *make* shall behave as if the −**i** option had been specified and errors from all commands associated with all targets shall be ignored.

.POSIX The application shall ensure that this special target is specified without prerequisites or commands. If it appears as the first non-comment line in the makefile, *make* shall process the makefile as specified by this section; otherwise, the behavior of *make* is unspecified.

.PRECIOUS Prerequisites of this special target shall not be removed if *make* receives one of the asynchronous events explicitly described in the ASYNCHRONOUS EVENTS section. Subsequent occurrences of **.PRECIOUS** shall add to the list of precious files. If no prerequisites are specified, all targets in the makefile shall be treated as if specified with **.PRECIOUS**.

XSI **.SCCS_GET** The application shall ensure that this special target is specified without prerequisites. If this special target is included in a makefile, the commands specified with this target shall replace the default commands associated with this special target (see **Default Rules** (on page 2819)). The commands specified with this target are used to get all SCCS files that are not found in the current directory.

When source files are named in a dependency list, *make* shall treat them just like any other target. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile. When a target has no dependencies, but is present in the directory, *make* shall assume that that file is up-to-date. If, however, an SCCS file named **SCCS/s.***source_file* is found for a target *source_file*, *make* compares the timestamp of the target file with that of the **SCCS/s.source_file** to assure the target is up-to-date. If the target is missing, or if the SCCS file is newer, *make* shall automatically issue the commands specified for the **.SCCS_GET** special target to retrieve the most recent version. However, if the target is writable by anyone, *make* shall not retrieve a new version.

.SILENT Prerequisites of this special target are targets themselves; this shall cause commands associated with them to not be written to the standard output before they are executed. Subsequent occurrences of **.SILENT** shall add to the list of targets with silent commands. If no prerequisites are specified, *make* shall behave as if the −**s** option had been specified and no commands or touch messages associated with any target shall be written to standard output.

.SUFFIXES Prerequisites of **.SUFFIXES** shall be appended to the list of known suffixes and are used in conjunction with the inference rules (see **Inference Rules** (on page 2816)). If **.SUFFIXES** does not have any prerequisites, the list of known suffixes shall be cleared.

The special targets **.IGNORE**, **.POSIX**, **.PRECIOUS**, **.SILENT**, and **.SUFFIXES** shall be specified without commands.

Targets with names consisting of a leading period followed by the uppercase letters `"POSIX"` and then any other characters are reserved for future standardization. Targets with names consisting of a leading period followed by one or more uppercase letters are reserved for implementation extensions.

Macros

Macro definitions are in the form:

`string1 = [string2]`

The macro named *string1* is defined as having the value of *string2*, where *string2* is defined as all characters, if any, after the equal sign, up to a comment character (`'#'`) or an unescaped <newline>. Any <blank>s immediately before or after the equal sign shall be ignored.

Applications shall select macro names from the set of characters consisting solely of periods, underscores, digits, and alphabetics from the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set). A macro name shall not contain an equals sign. Implementations may allow other characters in macro names as extensions.

Macros can appear anywhere in the makefile. Macro expansions using the forms $(*string1*) or ${*string1*} shall be replaced by *string2*, as follows:

- Macros in target lines shall be evaluated when the target line is read.
- Macros in makefile command lines shall be evaluated when the command is executed.
- Macros in the string before the equals sign in a macro definition shall be evaluated when the macro assignment is made.
- Macros after the equals sign in a macro definition shall not be evaluated until the defined macro is used in a rule or command, or before the equals sign in a macro definition.

The parentheses or braces are optional if *string1* is a single character. The macro $$ shall be replaced by the single character `'$'`. If *string1* in a macro expansion contains a macro expansion, the results are unspecified.

Macro expansions using the forms $(*string1*[:*subst1*=[*subst2*]]) or ${*string1*[:*subst1*=[*subst2*]]} can be used to replace all occurrences of *subst1* with *subst2* when the macro substitution is performed. The *subst1* to be replaced shall be recognized when it is a suffix at the end of a word in *string1* (where a *word*, in this context, is defined to be a string delimited by the beginning of the line, a <blank> or <newline>). If *string1* in a macro expansion contains a macro expansion, the results are unspecified.

Macro expansions in *string1* of macro definition lines shall be evaluated when read. Macro expansions in *string2* of macro definition lines shall be performed when the macro identified by *string1* is expanded in a rule or command.

Macro definitions shall be taken from the following sources, in the following logical order, before the makefile(s) are read.

1. Macros specified on the *make* utility command line, in the order specified on the command line. It is unspecified whether the internal macros defined in **Internal Macros** (on page 2818) are accepted from this source.
2. Macros defined by the *MAKEFLAGS* environment variable, in the order specified in the environment variable. It is unspecified whether the internal macros defined in **Internal Macros** (on page 2818) are accepted from this source.

3. The contents of the environment, excluding the *MAKEFLAGS* and *SHELL* variables and including the variables with null values.

4. Macros defined in the inference rules built into *make*.

Macro definitions from these sources shall not override macro definitions from a lower-numbered source. Macro definitions from a single source (for example, the *make* utility command line, the *MAKEFLAGS* environment variable, or the other environment variables) shall override previous macro definitions from the same source.

Macros defined in the makefile(s) shall override macro definitions that occur before them in the makefile(s) and macro definitions from source 4. If the −**e** option is not specified, macros defined in the makefile(s) shall override macro definitions from source 3. Macros defined in the makefile(s) shall not override macro definitions from source 1 or source 2.

Before the makefile(s) are read, all of the *make* utility command line options (except −**f** and −**p**) and *make* utility command line macro definitions (except any for the *MAKEFLAGS* macro), not already included in the *MAKEFLAGS* macro, shall be added to the *MAKEFLAGS* macro, quoted in an implementation-defined manner such that when *MAKEFLAGS* is read by another instance of the *make* command, the original macro's value is recovered. Other implementation-defined options and macros may also be added to the *MAKEFLAGS* macro. If this modifies the value of the *MAKEFLAGS* macro, or, if the *MAKEFLAGS* macro is modified at any subsequent time, the *MAKEFLAGS* environment variable shall be modified to match the new value of the *MAKEFLAGS* macro. The result of setting *MAKEFLAGS* in the Makefile is unspecified.

Before the makefile(s) are read, all of the *make* utility command line macro definitions (except the *MAKEFLAGS* macro or the *SHELL* macro) shall be added to the environment of *make*. Other implementation-defined variables may also be added to the environment of *make*.

The **SHELL** macro shall be treated specially. It shall be provided by *make* and set to the pathname of the shell command language interpreter (see *sh* (on page 3048)). The *SHELL* environment variable shall not affect the value of the **SHELL** macro. If **SHELL** is defined in the makefile or is specified on the command line, it shall replace the original value of the **SHELL** macro, but shall not affect the *SHELL* environment variable. Other effects of defining **SHELL** in the makefile or on the command line are implementation-defined.

Inference Rules

Inference rules are formatted as follows:

```
target:
<tab>command
[<tab>command]
...

line that does not begin with <tab> or #
```

The application shall ensure that the *target* portion is a valid target name (see **Target Rules** (on page 2813)) of the form **.s2** or **.s1.s2** (where **.s1** and **.s2** are suffixes that have been given as prerequisites of the **.SUFFIXES** special target and *s1* and *s2* do not contain any slashes or periods.) If there is only one period in the target, it is a single-suffix inference rule. Targets with two periods are double-suffix inference rules. Inference rules can have only one target before the colon.

The application shall ensure that the makefile does not specify prerequisites for inference rules; no characters other than white space shall follow the colon in the first line, except when creating the *empty rule*, described below. Prerequisites are inferred, as described below.

Inference rules can be redefined. A target that matches an existing inference rule shall overwrite the old inference rule. An empty rule can be created with a command consisting of simply a semicolon (that is, the rule still exists and is found during inference rule search, but since it is empty, execution has no effect). The empty rule also can be formatted as follows:

```
rule: ;
```

where zero or more <blank>s separate the colon and semicolon.

The *make* utility uses the suffixes of targets and their prerequisites to infer how a target can be made up-to-date. A list of inference rules defines the commands to be executed. By default, *make* contains a built-in set of inference rules. Additional rules can be specified in the makefile.

The special target **.SUFFIXES** contains as its prerequisites a list of suffixes that shall be used by the inference rules. The order in which the suffixes are specified defines the order in which the inference rules for the suffixes are used. New suffixes shall be appended to the current list by specifying a **.SUFFIXES** special target in the makefile. A **.SUFFIXES** target with no prerequisites shall clear the list of suffixes. An empty **.SUFFIXES** target followed by a new **.SUFFIXES** list is required to change the order of the suffixes.

Normally, the user would provide an inference rule for each suffix. The inference rule to update a target with a suffix **.s1** from a prerequisite with a suffix **.s2** is specified as a target **.s2.s1**. The internal macros provide the means to specify general inference rules (see **Internal Macros** (on page 2818)).

When no target rule is found to update a target, the inference rules shall be checked. The suffix of the target (**.s1**) to be built is compared to the list of suffixes specified by the **.SUFFIXES** special targets. If the **.s1** suffix is found in **.SUFFIXES**, the inference rules shall be searched in the order defined for the first **.s2.s1** rule whose prerequisite file (**$*.s2**) exists. If the target is out-of-date with respect to this prerequisite, the commands for that inference rule shall be executed.

If the target to be built does not contain a suffix and there is no rule for the target, the single suffix inference rules shall be checked. The single-suffix inference rules define how to build a target if a file is found with a name that matches the target name with one of the single suffixes appended. A rule with one suffix **.s2** is the definition of how to build *target* from **target.s2**. The other suffix (**.s1**) is treated as null.

XSI A tilde (`'~'`) in the above rules refers to an SCCS file in the current directory. Thus, the rule **.c~.o** would transform an SCCS C-language source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix, it is incompatible with *make*'s suffix point of view. Hence, the `'~'` is a way of changing any file reference into an SCCS file reference.

Libraries

If a target or prerequisite contains parentheses, it shall be treated as a member of an archive library. For the *lib*(*member*.**o**) expression *lib* refers to the name of the archive library and *member*.**o** to the member name. The application shall ensure that the member is an object file with the **.o** suffix. The modification time of the expression is the modification time for the member as kept in the archive library; see *ar* (on page 2336). The **.a** suffix shall refer to an archive library. The **.s2.a** rule shall be used to update a member in the library from a file with a suffix **.s2**.

Internal Macros

The *make* utility shall maintain five internal macros that can be used in target and inference rules. In order to clearly define the meaning of these macros, some clarification of the terms *target rule*, *inference rule*, *target*, and *prerequisite* is necessary.

Target rules are specified by the user in a makefile for a particular target. Inference rules are user-specified or *make*-specified rules for a particular class of target name. Explicit prerequisites are those prerequisites specified in a makefile on target lines. Implicit prerequisites are those prerequisites that are generated when inference rules are used. Inference rules are applied to implicit prerequisites or to explicit prerequisites that do not have target rules defined for them in the makefile. Target rules are applied to targets specified in the makefile.

Before any target in the makefile is updated, each of its prerequisites (both explicit and implicit) shall be updated. This shall be accomplished by recursively processing each prerequisite. Upon recursion, each prerequisite shall become a target itself. Its prerequisites in turn shall be processed recursively until a target is found that has no prerequisites, at which point the recursion stops. The recursion then shall back up, updating each target as it goes.

In the definitions that follow, the word *target* refers to one of:

- A target specified in the makefile
- An explicit prerequisite specified in the makefile that becomes the target when *make* processes it during recursion
- An implicit prerequisite that becomes a target when *make* processes it during recursion

In the definitions that follow, the word *prerequisite* refers to one of the following:

- An explicit prerequisite specified in the makefile for a particular target
- An implicit prerequisite generated as a result of locating an appropriate inference rule and corresponding file that matches the suffix of the target

The five internal macros are:

$@ The $@ shall evaluate to the full target name of the current target, or the archive filename part of a library archive target. It shall be evaluated for both target and inference rules.

For example, in the **.c.a** inference rule, $@ represents the out-of-date **.a** file to be built. Similarly, in a makefile target rule to build **lib.a** from **file.c**, $@ represents the out-of-date **lib.a**.

$% The $% macro shall be evaluated only when the current target is an archive library member of the form *libname*(*member*.**o**). In these cases, $@ shall evaluate to *libname* and $% shall evaluates to *member*.**o**. The $% macro shall be evaluated for both target and inference rules.

For example, in a makefile target rule to build **lib.a**(**file.o**), $% represents **file.o**, as opposed to $@, which represents **lib.a**.

$? The $? macro shall evaluate to the list of prerequisites that are newer than the current target. It shall be evaluated for both target and inference rules.

For example, in a makefile target rule to build *prog* from **file1.o**, **file2.o**, and **file3.o**, and where *prog* is not out of date with respect to **file1.o**, but is out of date with respect to **file2.o** and **file3.o**, $? represents **file2.o** and **file3.o**.

$<	In an inference rule, the $< macro shall evaluate to the filename whose existence allowed the inference rule to be chosen for the target. In the **.DEFAULT** rule, the $< macro shall evaluate to the current target name. The meaning of the $< macro shall be otherwise unspecified. For example, in the **.c.a** inference rule, $< represents the prerequisite **.c** file.
$*	The $* macro shall evaluate to the current target name with its suffix deleted. It shall be evaluated at least for inference rules. For example, in the **.c.a** inference rule, $*.o represents the out-of-date **.o** file that corresponds to the prerequisite **.c** file.

Each of the internal macros has an alternative form. When an uppercase `'D'` or `'F'` is appended to any of the macros, the meaning shall be changed to the *directory part* for `'D'` and *filename part* for `'F'`. The directory part is the path prefix of the file without a trailing slash; for the current directory, the directory part is `'.'`. When the $? macro contains more than one prerequisite filename, the $(?D) and $(?F) (or ${?D} and ${?F}) macros expand to a list of directory name parts and filename parts respectively.

For the target *lib*(*member*.**o**) and the **s2.a** rule, the internal macros shall be defined as:

$<	*member*.**s2**
$*	*member*
$@	*lib*
$?	*member*.**s2**
$%	*member*.**o**

Default Rules

The default rules for *make* shall achieve results that are the same as if the following were used. Implementations that do not support the C-Language Development Utilities option may omit **CC**, **CFLAGS**, **YACC**, **YFLAGS**, **LEX**, **LFLAGS**, **LDFLAGS**, and the **.c**, **.y**, and **.l** inference rules. Implementations that do not support FORTRAN may omit **FC**, **FFLAGS**, and the **.f** inference rules. Implementations may provide additional macros and rules.

```
SPECIAL TARGETS

.SCCS_GET: sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@


.SUFFIXES: .o .c .y .l .a .sh .f .c˜ .y˜ .l˜ .sh˜ .f˜

MACROS

MAKE=make
AR=ar
ARFLAGS=−rv
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LDFLAGS=
CC=c99
CFLAGS=−O
FC=fort77
```

```
FFLAGS=-O 1
GET=get
GFLAGS=
SCCSFLAGS=
SCCSGETFLAGS=-s


SINGLE SUFFIX RULES

.c:
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<

.f:
    $(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<

.sh:
    cp $< $@
    chmod a+x $@

.c~:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c

.f~:
    $(GET) $(GFLAGS) -p $< > $*.f
    $(FC) $(FFLAGS) $(LDFLAGS) -o $@ $*.f

.sh~:
    $(GET) $(GFLAGS) -p $< > $*.sh
    cp $*.sh $@
    chmod a+x $@


DOUBLE SUFFIX RULES

.c.o:
    $(CC) $(CFLAGS) -c $<

.f.o:
    $(FC) $(FFLAGS) -c $<

.y.o:
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm -f y.tab.c
    mv y.tab.o $@

.l.o:
    $(LEX) $(LFLAGS) $<
    $(CC) $(CFLAGS) -c lex.yy.c
    rm -f lex.yy.c
    mv lex.yy.o $@

.y.c:
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@

.l.c:
    $(LEX) $(LFLAGS) $<
```

```
    mv lex.yy.c $@
```

XSI

```
.c~.o:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) -c $*.c

.f~.o:
    $(GET) $(GFLAGS) -p $< > $*.f
    $(FC) $(FFLAGS) -c $*.f

.y~.o:
    $(GET) $(GFLAGS) -p $< > $*.y
    $(YACC) $(YFLAGS) $*.y
    $(CC) $(CFLAGS) -c y.tab.c
    rm -f y.tab.c
    mv y.tab.o $@

.l~.o:
    $(GET) $(GFLAGS) -p $< > $*.l
    $(LEX) $(LFLAGS) $*.l
    $(CC) $(CFLAGS) -c lex.yy.c
    rm -f lex.yy.c
    mv lex.yy.o $@

.y~.c:
    $(GET) $(GFLAGS) -p $< > $*.y
    $(YACC) $(YFLAGS) $*.y
    mv y.tab.c $@

.l~.c:
    $(GET) $(GFLAGS) -p $< > $*.l
    $(LEX) $(LFLAGS) $*.l
    mv lex.yy.c $@
```

```
.c.a:
    $(CC) -c $(CFLAGS) $<
    $(AR) $(ARFLAGS) $@ $*.o
    rm -f $*.o

.f.a:
    $(FC) -c $(FFLAGS) $<
    $(AR) $(ARFLAGS) $@ $*.o
    rm -f $*.o
```

EXIT STATUS

When the −**q** option is specified, the *make* utility shall exit with one of the following values:

0	Successful completion.
1	The target was not up-to-date.
>1	An error occurred.

When the −**q** option is not specified, the *make* utility shall exit with one of the following values:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If there is a source file (such as **./source.c**) and there are two SCCS files corresponding to it (**./s.source.c** and **./SCCS/s.source.c**), on XSI-conformant systems *make* uses the SCCS file in the current directory. However, users are advised to use the underlying SCCS utilities (*admin*, *delta*, *get*, and so on) or the *sccs* utility for all source files in a given directory. If both forms are used for a given source file, future developers are very likely to be confused.

It is incumbent upon portable makefiles to specify the **.POSIX** special target in order to guarantee that they are not affected by local extensions.

The −**k** and −**S** options are both present so that the relationship between the command line, the *MAKEFLAGS* variable, and the makefile can be controlled precisely. If the **k** flag is passed in *MAKEFLAGS* and a command is of the form:

```
$(MAKE) −S foo
```

then the default behavior is restored for the child *make*.

When the −**n** option is specified, it is always added to *MAKEFLAGS*. This allows a recursive *make* −**n** *target* to be used to see all of the action that would be taken to update *target*.

Because of widespread historical practice, interpreting a ’#’ number sign inside a variable as the start of a comment has the unfortunate side effect of making it impossible to place a number sign in a variable, thus forbidding something like:

```
CFLAGS = "−D COMMENT_CHAR=’#’"
```

Many historical *make* utilities stop chaining together inference rules when an intermediate target is nonexistent. For example, it might be possible for a *make* to determine that both **.y.c** and **.c.o** could be used to convert a **.y** to a **.o**. Instead, in this case, *make* requires the use of a **.y.o** rule.

The best way to provide portable makefiles is to include all of the rules needed in the makefile itself. The rules provided use only features provided by other parts of this volume of IEEE Std 1003.1-200x. The default rules include rules for optional commands in this volume of IEEE Std 1003.1-200x. Only rules pertaining to commands that are provided are needed in an implementation's default set.

Macros used within other macros are evaluated when the new macro is used rather than when the new macro is defined. Therefore:

```
MACRO = value1
NEW   = $(MACRO)
MACRO = value2

target:
    echo $(NEW)
```

would produce *value2* and not *value1* since **NEW** was not expanded until it was needed in the *echo* command line.

Some historical applications have been known to intermix *target_name* and *macro=name* operands on the command line, expecting that all of the macros are processed before any of the targets are dealt with. Conforming applications do not do this, although some backward compatibility support may be included in some implementations.

The following characters in filenames may give trouble: ’=’, ’:’, ’‘’, ’’’, and ’@’. For inference rules, the description of $< and $? seem similar. However, an example shows the minor difference. In a makefile containing:

```
foo.o: foo.h
```

if **foo.h** is newer than **foo.o**, yet **foo.c** is older than **foo.o**, the built-in rule to make **foo.o** from **foo.c** is used, with $< equal to **foo.c** and $? equal to **foo.h**. If **foo.c** is also newer than **foo.o**, $< is equal to **foo.c** and $? is equal to **foo.h foo.c**.

EXAMPLES

1. The following command:

   ```
   make
   ```

   makes the first target found in the makefile.

2. The following command:

   ```
   make junk
   ```

   makes the target **junk**.

3. The following makefile says that **pgm** depends on two files, **a.o** and **b.o**, and that they in turn depend on their corresponding source files (**a.c** and **b.c**), and a common file **incl.h**:

   ```
   pgm: a.o b.o
       c99 a.o b.o -o pgm
   a.o: incl.h a.c
       c99 -c a.c
   b.o: incl.h b.c
       c99 -c b.c
   ```

4. An example for making optimized **.o** files from **.c** files is:

   ```
   .c.o:
       c99 -c -O $*.c
   ```

   or:

   ```
   .c.o:
       c99 -c -O $<
   ```

5. The most common use of the archive interface follows. Here, it is assumed that the source files are all C-language source:

   ```
   lib: lib(file1.o) lib(file2.o) lib(file3.o)
       @echo lib is now up-to-date
   ```

   The **.c.a** rule is used to make **file1.o**, **file2.o**, and **file3.o** and insert them into **lib**.

   The treatment of escaped <newline>s throughout the makefile is historical practice. For example, the inference rule:

   ```
   .c.o\
   :
   ```

   works, and the macro:

   ```
   f=  bar baz\
       biz
   a:
       echo ==$f==
   ```

echoes `"==bar baz biz=="`.

If $? were:

```
/usr/include/stdio.h /usr/include/unistd.h foo.h
```

then $(?D) would be:

```
/usr/include /usr/include .
```

and $(?F) would be:

```
stdio.h unistd.h foo.h
```

6. The contents of the built-in rules can be viewed by running:

```
make −p −f /dev/null 2>/dev/null
```

RATIONALE

The *make* utility described in this volume of IEEE Std 1003.1-200x is intended to provide the means for changing portable source code into executables that can be run on a IEEE Std 1003.1-200x-conforming system. It reflects the most common features present in System V and BSD *make*s.

Historically, the *make* utility has been an especially fertile ground for vendor and research organization-specific syntax modifications and extensions. Examples include:

- Syntax supporting parallel execution (such as from various multi-processor vendors, GNU, and others)
- Additional ''operators'' separating targets and their prerequisites (System V, BSD, and others)
- Specifying that command lines containing the strings `"${MAKE}"` and `"$(MAKE)"` are executed when the −**n** option is specified (GNU and System V)
- Modifications of the meaning of internal macros when referencing libraries (BSD and others)
- Using a single instance of the shell for all of the command lines of the target (BSD and others)
- Allowing spaces as well as tabs to delimit command lines (BSD)
- Adding C preprocessor-style ''include'' and ''ifdef'' constructs (System V, GNU, BSD, and others)
- Remote execution of command lines (Sprite and others)
- Specifying additional special targets (BSD, System V, and most others)

Additionally, many vendors and research organizations have rethought the basic concepts of *make*, creating vastly extended, as well as completely new, syntaxes. Each of these versions of *make* fulfills the needs of a different community of users; it is unreasonable for this volume of IEEE Std 1003.1-200x to require behavior that would be incompatible (and probably inferior) to historical practice for such a community.

In similar circumstances, when the industry has enough sufficiently incompatible formats as to make them irreconcilable, this volume of IEEE Std 1003.1-200x has followed one or both of two courses of action. Commands have been renamed (*cksum*, *echo*, and *pax*) and/or command line options have been provided to select the desired behavior (*grep*, *od*, and *pax*).

Because the syntax specified for the *make* utility is, by and large, a subset of the syntaxes accepted by almost all versions of *make*, it was decided that it would be counter-productive to change the name. And since the makefile itself is a basic unit of portability, it would not be completely effective to reserve a new option letter, such as *make* −**P**, to achieve the portable behavior. Therefore, the special target **.POSIX** was added to the makefile, allowing users to specify ''standard'' behavior. This special target does not preclude extensions in the *make* utility, nor does it preclude such extensions being used by the makefile specifying the target; it does, however, preclude any extensions from being applied that could alter the behavior of previously valid syntax; such extensions must be controlled via command line options or new special targets. It is incumbent upon portable makefiles to specify the **.POSIX** special target in order to guarantee that they are not affected by local extensions.

The portable version of *make* described in this reference page is not intended to be the state-of-the-art software generation tool and, as such, some newer and more leading-edge features have not been included. An attempt has been made to describe the portable makefile in a manner that does not preclude such extensions as long as they do not disturb the portable behavior described here.

When the −**n** option is specified, it is always added to *MAKEFLAGS*. This allows a recursive *make* −**n** *target* to be used to see all of the action that would be taken to update *target*.

The definition of *MAKEFLAGS* allows both the System V letter string and the BSD command line formats. The two formats are sufficiently different to allow implementations to support both without ambiguity.

Early proposals stated that an ''unquoted'' number sign was treated as the start of a comment. The *make* utility does not pay any attention to quotes. A number sign starts a comment regardless of its surroundings.

The text about ''other implementation-defined pathnames may also be tried'' in addition to **./makefile** and **./Makefile** is to allow such extensions as **SCCS/s.Makefile** and other variations. It was made an implementation-defined requirement (as opposed to unspecified behavior) to highlight surprising implementations that might select something unexpected like **/etc/Makefile**. XSI-conformant systems also try **./s.makefile**, **SCCS/s.makefile**, **./s.Makefile**, and **SCCS/s.Makefile**.

Early proposals contained the macro **NPROC** as a means of specifying that *make* should use *n* processes to do the work required. While this feature is a valuable extension for many systems, it is not common usage and could require other non-trivial extensions to makefile syntax. This extension is not required by this volume of IEEE Std 1003.1-200x, but could be provided as a compatible extension. The macro **PARALLEL** is used by some historical systems with essentially the same meaning (but without using a name that is a common system limit value). It is suggested that implementors recognize the existing use of **NPROC** and/or **PARALLEL** as extensions to *make*.

The default rules are based on System V. The default **CC=** value is *c99* instead of *cc* because this volume of IEEE Std 1003.1-200x does not standardize the utility named *cc*. Thus, every conforming application would be required to define **CC=***c99* to expect to run. There is no advantage conferred by the hope that the makefile might hit the ''preferred'' compiler because this cannot be guaranteed to work. Also, since the portable makescript can only use the *c99* options, no advantage is conferred in terms of what the script can do. It is a quality-of-implementation issue as to whether *c99* is as valuable as *cc*.

The −**d** option to *make* is frequently used to produce debugging information, but is too implementation-defined to add to this volume of IEEE Std 1003.1-200x.

The −**p** option is not passed in *MAKEFLAGS* on most historical implementations and to change this would cause many implementations to break without sufficiently increased portability.

Commands that begin with a plus sign (`'+'`) are executed even if the −**n** option is present. Based on the GNU version of *make*, the behavior of −**n** when the plus-sign prefix is encountered has been extended to apply to −**q** and −**t** as well. However, the System V convention of forcing command execution with −**n** when the command line of a target contains either of the strings `"$(MAKE)"` or `"${MAKE}"` has not been adopted. This functionality appeared in early proposals, but the danger of this approach was pointed out with the following example of a portion of a makefile:

```
subdir:
    cd subdir; rm all_the_files; $(MAKE)
```

The loss of the System V behavior in this case is well-balanced by the safety afforded to other makefiles that were not aware of this situation. In any event, the command line plus-sign prefix can provide the desired functionality.

The double colon in the target rule format is supported in BSD systems to allow more than one target line containing the same target name to have commands associated with it. Since this is not functionality described in the SVID or XPG3 it has been allowed as an extension, but not mandated.

The default rules are provided with text specifying that the built-in rules shall be the same *as if* the listed set were used. The intent is that implementations should be able to use the rules without change, but will be allowed to alter them in ways that do not affect the primary behavior.

The best way to provide portable makefiles is to include all of the rules needed in the makefile itself. The rules provided use only features provided by other portions of this volume of IEEE Std 1003.1-200x. The default rules include rules for optional commands in this volume of IEEE Std 1003.1-200x. Only rules pertaining to commands that are provided are needed in the default set of an implementation.

One point of discussion was whether to drop the default rules list from this volume of IEEE Std 1003.1-200x. They provide convenience, but do not enhance portability of applications. The prime benefit is in portability of users who wish to type *make command* and have the command build from a **command.c** file.

The historical *MAKESHELL* feature was omitted. In some implementations it is used to let a user override the shell to be used to run *make* commands. This was confusing; for a portable *make*, the shell should be chosen by the makefile writer or specified on the *make* command line and not by a user running *make*.

The *make* utilities in most historical implementations process the prerequisites of a target in left-to-right order, and the makefile format requires this. It supports the standard idiom used in many makefiles that produce *yacc* programs; for example:

```
foo: y.tab.o lex.o main.o
    $(CC) $(CFLAGS) −o $@ t.tab.o lex.o main.o
```

In this example, if *make* chose any arbitrary order, the **lex.o** might not be made with the correct **y.tab.h**. Although there may be better ways to express this relationship, it is widely used historically. Implementations that desire to update prerequisites in parallel should require an explicit extension to *make* or the makefile format to accomplish it, as described previously.

The algorithm for determining a new entry for target rules is partially unspecified. Some historical *make*s allow blank, empty, or comment lines within the collection of commands marked by leading <tab>s. A conforming makefile must ensure that each command starts with a <tab>, but implementations are free to ignore blank, empty, and comment lines without triggering the start of a new entry.

The ASYNCHRONOUS EVENTS section includes having SIGTERM and SIGHUP, along with the more traditional SIGINT and SIGQUIT, remove the current target unless directed not to do so. SIGTERM and SIGHUP were added to parallel other utilities that have historically cleaned up their work as a result of these signals. When *make* receives any signal other than SIGQUIT, it is required to resend itself the signal it received so that it exits with a status that reflects the signal. The results from SIGQUIT are partially unspecified because, on systems that create **core** files upon receipt of SIGQUIT, the **core** from *make* would conflict with a core file from the command that was running when the SIGQUIT arrived. The main concern was to prevent damaged files from appearing up-to-date when *make* is rerun.

The **.PRECIOUS** special target was extended to affect all targets globally (by specifying no prerequisites). The **.IGNORE** and **.SILENT** special targets were extended to allow prerequisites; it was judged to be more useful in some cases to be able to turn off errors or echoing for a list of targets than for the entire makefile. These extensions to the *make* in System V were made to match historical practice from the BSD *make*.

Macros are not exported to the environment of commands to be run. This was never the case in any historical *make* and would have serious consequences. The environment is the same as the environment to *make* except that *MAKEFLAGS* and macros defined on the *make* command line are added.

Some implementations do not use *system*( ) for all command lines, as required by the portable makefile format; as a performance enhancement, they select lines without shell metacharacters for direct execution by *execve*( ). There is no requirement that *system*( ) be used specifically, but merely that the same results be achieved. The metacharacters typically used to bypass the direct *execve*( ) execution have been any of:

`= | ˆ ( ) ; & < > * ? [ ] : $ ‘ ’ " \ \n`

The default in some advanced versions of *make* is to group all the command lines for a target and execute them using a single shell invocation; the System V method is to pass each line individually to a separate shell. The single-shell method has the advantages in performance and the lack of a requirement for many continued lines. However, converting to this newer method has caused portability problems with many historical makefiles, so the behavior with the POSIX makefile is specified to be the same as that of System V. It is suggested that the special target **.ONESHELL** be used as an implementation extension to achieve the single-shell grouping for a target or group of targets.

Novice users of *make* have had difficulty with the historical need to start commands with a <tab>. Since it is often difficult to discern differences between <tab>s and <space>s on terminals or printed listings, confusing bugs can arise. In early proposals, an attempt was made to correct this problem by allowing leading <blank>s instead of <tab>s. However, implementors reported many makefiles that failed in subtle ways following this change, and it is difficult to implement a *make* that unambiguously can differentiate between macro and command lines. There is extensive historical practice of allowing leading spaces before macro definitions. Forcing macro lines into column 1 would be a significant backwards-compatibility problem for some makefiles. Therefore, historical practice was restored.

The System V INCLUDE feature was considered, but not included. This would treat a line that began in the first column and contained INCLUDE <*filename*> as an indication to read <*filename*> at that point in the makefile. This is difficult to use in a portable way, and it raises concerns about nesting levels and diagnostics. System V, BSD, GNU, and others have used different methods for including files.

The System V dynamic dependency feature was not included. It would support:

```
cat: $$@.c
```

that would expand to;

```
cat: cat.c
```

This feature exists only in the new version of System V *make* and, while useful, is not in wide usage. This means that macros are expanded twice for prerequisites: once at makefile parse time and once at target update time.

Consideration was given to adding metarules to the POSIX *make*. This would make **%.o: %.c** the same as **.c.o:**. This is quite useful and available from some vendors, but it would cause too many changes to this *make* to support. It would have introduced rule chaining and new substitution rules. However, the rules for target names have been set to reserve the `'%'` and `'"'` characters. These are traditionally used to implement metarules and quoting of target names, respectively. Implementors are strongly encouraged to use these characters only for these purposes.

A request was made to extend the suffix delimiter character from a period to any character. The metarules feature in newer *make*s solves this problem in a more general way. This volume of IEEE Std 1003.1-200x is staying with the more conservative historical definition.

The standard output format for the **−p** option is not described because it is primarily a debugging option and because the format is not generally useful to programs. In historical implementations the output is not suitable for use in generating makefiles. The **−p** format has been variable across historical implementations. Therefore, the definition of **−p** was only to provide a consistently named option for obtaining *make* script debugging information.

Some historical implementations have not cleared the suffix list with **−r**.

Implementations should be aware that some historical applications have intermixed *target_name* and *macro*=*value* operands on the command line, expecting that all of the macros are processed before any of the targets are dealt with. Conforming applications do not do this, but some backwards-compatibility support may be warranted.

Empty inference rules are specified with a semicolon command rather than omitting all commands, as described in an early proposal. The latter case has no traditional meaning and is reserved for implementation extensions, such as in GNU *make*.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *c99*, *get*, *lex*, *sh*, *yacc*, the System Interfaces volume of IEEE Std 1003.1-200x, *system*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the Software Development Utilities option.

The Open Group Corrigendum U029/1 is applied, correcting a typographical error in the SPECIAL TARGETS section.

In the ENVIRONMENT VARIABLES section, the *PROJECTDIR* description is updated from ‘‘otherwise, the home directory of a user of that name is examined’’ to ‘‘otherwise, the value of *PROJECTDIR* is treated as a user name and that user’s initial working directory is examined’’.

It is specified whether the command line is related to the makefile or to the *make* command, and the macro processing rules are updated to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

PASC Interpretation 1003.2 #193 is applied.

NAME

man — display system documentation

SYNOPSIS

```
man [−k] name...
```

DESCRIPTION

The *man* utility shall write information about each of the *name* operands. If *name* is the name of a standard utility, *man* at a minimum shall write a message describing the syntax used by the standard utility, its options, and operands. If more information is available, the *man* utility shall provide it in an implementation-defined manner.

An implementation may provide information for values of *name* other than the standard utilities. Standard utilities that are listed as optional and that are not supported by the implementation either shall cause a brief message indicating that fact to be displayed or shall cause a full display of information as described previously.

OPTIONS

The *man* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−k Interpret *name* operands as keywords to be used in searching a utilities summary database that contains a brief purpose entry for each standard utility and write lines from the summary database that match any of the keywords. The keyword search shall produce results that are the equivalent of the output of the following command:

```
grep −Ei '
name
name
...
' summary-database
```

This assumes that the *summary-database* is a text file with a single entry per line; this organization is not required and the example using *grep* **−Ei** is merely illustrative of the type of search intended. The purpose entry to be included in the database shall consist of a terse description of the purpose of the utility.

OPERANDS

The following operand shall be supported:

name A keyword or the name of a standard utility. When **−k** is not specified and *name* does not represent one of the standard utilities, the results are unspecified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *man*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and in the summary database). The value of *LC_CTYPE* need not affect the format of the information written about the name operands.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PAGER	Determine an output filtering command for writing the output to a terminal. Any string acceptable as a *command_string* operand to the *sh* −**c** command shall be valid. When standard output is a terminal device, the reference page output shall be piped through the command. If the *PAGER* variable is null or not set, the command shall be either *more* or another paginator utility documented in the system documentation.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *man* utility shall write text describing the syntax of the utility *name*, its options and its operands, or, when −**k** is specified, lines from the summary database. The format of this text is implementation-defined.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

It is recognized that the *man* utility is only of minimal usefulness as specified. The opinion of the standard developers was strongly divided as to how much or how little information *man* should be required to provide. They considered, however, that the provision of some portable way of accessing documentation would aid user portability. The arguments against a fuller specification were:

- Large quantities of documentation should not be required on a system that does not have excess disk space.
- The current manual system does not present information in a manner that greatly aids user portability.
- A ‘‘better help system’’ is currently an area in which vendors feel that they can add value to their POSIX implementations.

The −**f** option was considered, but due to implementation differences, it was not included in this volume of IEEE Std 1003.1-200x.

The description was changed to be more specific about what has to be displayed for a utility. The standard developers considered it insufficient to allow a display of only the synopsis without giving a short description of what each option and operand does.

The ‘‘purpose’’ entry to be included in the database can be similar to the section title (less the numeric prefix) from this volume of IEEE Std 1003.1-200x for each utility. These titles are similar to those used in historical systems for this purpose.

See *mailx* for rationale concerning the default paginator.

The caveat in the *LC_CTYPE* description was added because it is not a requirement that an implementation provide reference pages for all of its supported locales on each system; changing *LC_CTYPE* does not necessarily translate the reference page into another language. This is equivalent to the current state of *LC_MESSAGES* in IEEE Std 1003.1-200x—locale-specific messages are not yet a requirement.

The historical *MANPATH* variable is not included in POSIX because no attempt is made to specify naming conventions for reference page files, nor even to mandate that they are files at all. On some implementations they could be a true database, a hypertext file, or even fixed strings within the *man* executable. The standard developers considered the portability of reference pages to be outside their scope of work. However, users should be aware that *MANPATH* is implemented on a number of historical systems and that it can be used to tailor the search pattern for reference pages from the various categories (utilities, functions, file formats, and so on) when the system administrator reveals the location and conventions for reference pages on the system.

The keyword search can rely on at least the text of the section titles from these utility descriptions, and the implementation may add more keywords. The term ‘‘section titles’’ refers to the strings such as:

```
man − Display system documentation
ps − Report process status
```

FUTURE DIRECTIONS

None.

SEE ALSO

more

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

NAME

mesg — permit or deny messages

SYNOPSIS

UP
```
mesg [y|n]
```

DESCRIPTION

The *mesg* utility shall control whether other users are allowed to send messages via *write*, *talk*, or other utilities to a terminal device. The terminal device affected shall be determined by searching for the first terminal in the sequence of devices associated with standard input, standard output, and standard error, respectively. With no arguments, *mesg* shall report the current state without changing it. Processes with appropriate privileges may be able to send messages to the terminal independent of the current state.

OPTIONS

None.

OPERANDS

The following operands shall be supported in the POSIX locale:

`y`	Grant permission to other users to send messages to the terminal device.
`n`	Deny permission to other users to send messages to the terminal device.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *mesg*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written (by *mesg*) to standard error.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

If no operand is specified, *mesg* shall display the current terminal state in an unspecified format.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Receiving messages is allowed.
1	Receiving messages is not allowed.
>1	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The mechanism by which the message status of the terminal is changed is unspecified. Therefore, unspecified actions may cause the status of the terminal to change after *mesg* has successfully completed. These actions may include, but are not limited to: another invocation of the *mesg* utility, login procedures; invocation of the *stty* utility, invocation of the *chmod* utility or *chmod*( ) function, and so on.

EXAMPLES

None.

RATIONALE

The terminal changed by *mesg* is that associated with the standard input, output, or error, rather than the controlling terminal for the session. This is because users logged in more than once should be able to change any of their login terminals without having to stop the job running in those sessions. This is not a security problem involving the terminals of other users because appropriate privileges would be required to affect the terminal of another user.

The method of checking each of the first three file descriptors in sequence until a terminal is found was adopted from System V.

The file **/dev/tty** is not specified for the terminal device because it was thought to be too restrictive. Typical environment changes for the **n** operand are that write permissions are removed for *others* and *group* from the appropriate device. It was decided to leave the actual description of what is done as unspecified because of potential differences between implementations.

The format for standard output is unspecified because of differences between historical implementations. This output is generally not useful to shell scripts (they can use the exit status), so exact parsing of the output is unnecessary.

FUTURE DIRECTIONS

None.

SEE ALSO

talk, *write*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

NAME

mkdir — make directories

SYNOPSIS

```
mkdir [−p][−m mode] dir...
```

DESCRIPTION

The *mkdir* utility shall create the directories specified by the operands, in the order specified.

For each *dir* operand, the *mkdir* utility shall perform actions equivalent to the *mkdir*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called with the following arguments:

1. The *dir* operand is used as the *path* argument.
2. The value of the bitwise-inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO is used as the *mode* argument. (If the **−m** option is specified, the *mode* option-argument overrides this default.)

OPTIONS

The *mkdir* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−m *mode* Set the file permission bits of the newly-created directory to the specified *mode* value. The *mode* option-argument shall be the same as the *mode* operand defined for the *chmod* utility. In the *symbolic_mode* strings, the *op* characters `'+'` and `'−'` shall be interpreted relative to an assumed initial mode of *a*=*rwx*; `'+'` shall add permissions to the default mode, `'−'` shall delete permissions from the default mode.

−p Create any missing intermediate pathname components.

For each *dir* operand that does not name an existing directory, effects equivalent to those caused by the following command shall occur:

```
mkdir −p −m $(umask −S),u+wx $(dirname dir) &&
mkdir [−m mode] dir
```

where the **−m** *mode* option represents that option supplied to the original invocation of *mkdir*, if any.

Each *dir* operand that names an existing directory shall be ignored without error.

OPERANDS

The following operand shall be supported:

dir A pathname of a directory to be created.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *mkdir*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All the specified directories were created successfully or the −**p** option was specified and all the specified directories now exist.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The default file mode for directories is *a*=*rwx* (777 on most systems) with selected permissions removed in accordance with the file mode creation mask. For intermediate pathname components created by *mkdir*, the mode is the default modified by *u*+*wx* so that the subdirectories can always be created regardless of the file mode creation mask; if different ultimate permissions are desired for the intermediate directories, they can be changed afterwards with *chmod*.

Note that some of the requested directories may have been created even if an error occurs.

EXAMPLES

None.

RATIONALE

The System V −**m** option was included to control the file mode.

The System V −**p** option was included to create any needed intermediate directories and to complement the functionality provided by *rmdir* for removing directories in the path prefix as they become empty. Because no error is produced if any path component already exists, the −**p** option is also useful to ensure that a particular directory exists.

The functionality of *mkdir* is described substantially through a reference to the *mkdir*( ) function in the System Interfaces volume of IEEE Std 1003.1-200x. For example, by default, the mode of the directory is affected by the file mode creation mask in accordance with the specified behavior of the *mkdir*( ) function. In this way, there is less duplication of effort required for describing details of the directory creation.

FUTURE DIRECTIONS

None.

SEE ALSO

rm, *rmdir*, *umask*, the System Interfaces volume of IEEE Std 1003.1-200x, *mkdir*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

NAME

mkfifo — make FIFO special files

SYNOPSIS

```
mkfifo [−m mode] file...
```

DESCRIPTION

The *mkfifo* utility shall create the FIFO special files specified by the operands, in the order specified.

For each *file* operand, the *mkfifo* utility shall perform actions equivalent to the *mkfifo*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called with the following arguments:

1. The *file* operand is used as the *path* argument.
2. The value of the bitwise-inclusive OR of S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH is used as the *mode* argument. (If the −**m** option is specified, the value of the *mkfifo*( ) *mode* argument is unspecified, but the FIFO shall at no time have permissions less restrictive than the −**m** *mode* option-argument.)

OPTIONS

The *mkfifo* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−**m** *mode*	Set the file permission bits of the newly-created FIFO to the specified *mode* value. The *mode* option-argument shall be the same as the *mode* operand defined for the *chmod* utility. In the *symbolic_mode* strings, the *op* characters `'+'` and `'−'` shall be interpreted relative to an assumed initial mode of *a*=*rw*.

OPERANDS

The following operand shall be supported:

file	A pathname of the FIFO special file to be created.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *mkfifo*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All the specified FIFO special files were created successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

This utility was added to permit shell applications to create FIFO special files.

The −**m** option was added to control the file mode, for consistency with the similar functionality provided the *mkdir* utility.

Early proposals included a −**p** option similar to the *mkdir* −**p** option that created intermediate directories leading up to the FIFO specified by the final component. This was removed because it is not commonly needed and is not common practice with similar utilities.

The functionality of *mkfifo* is described substantially through a reference to the *mkfifo*( ) function in the System Interfaces volume of IEEE Std 1003.1-200x. For example, by default, the mode of the FIFO file is affected by the file mode creation mask in accordance with the specified behavior of the *mkfifo*( ) function. In this way, there is less duplication of effort required for describing details of the file creation.

FUTURE DIRECTIONS

None.

SEE ALSO

umask, the System Interfaces volume of IEEE Std 1003.1-200x, *mkfifo*( )

CHANGE HISTORY

First released in Issue 3.

NAME

more — display files on a page-by-page basis

SYNOPSIS

UP
```
more [−ceisu][−n number][−p command][−t tagstring][file ...]
```

DESCRIPTION

The *more* utility shall read files and either write them to the terminal on a page-by-page basis or filter them to standard output. If standard output is not a terminal device, all input files shall be copied to standard output in their entirety, without modification, except as specified for the **−s** option. If standard output is a terminal device, the files shall be written a number of lines (one screenful) at a time under the control of user commands. See the EXTENDED DESCRIPTION section.

Certain block-mode terminals do not have all the capabilities necessary to support the complete *more* definition; they are incapable of accepting commands that are not terminated with a <newline>. Implementations that support such terminals shall provide an operating mode to *more* in which all commands can be terminated with a <newline> on those terminals. This mode:

- Shall be documented in the system documentation
- Shall, at invocation, inform the user of the terminal deficiency that requires the <newline> usage and provide instructions on how this warning can be suppressed in future invocations
- Shall not be required for implementations supporting only fully capable terminals
- Shall not affect commands already requiring <newline>s
- Shall not affect users on the capable terminals from using *more* as described in this volume of IEEE Std 1003.1-200x

OPTIONS

The *more* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	If a screen is to be written that has no lines in common with the current screen, or *more* is writing its first screen, *more* shall not scroll the screen, but instead shall redraw each line of the screen in turn, from the top of the screen to the bottom. In addition, if *more* is writing its first screen, the screen shall be cleared. This option may be silently ignored on devices with insufficient terminal capabilities.
−e	By default, *more* shall exit immediately after writing the last line of the last file in the argument list. If the **−e** option is specified: 1. If there is only a single file in the argument list and that file was completely displayed on a single screen, *more* shall exit immediately after writing the last line of that file. 2. Otherwise, *more* shall exit only after reaching end-of-file on the last file in the argument list twice without an intervening operation. See the EXTENDED DESCRIPTION section.
−i	Perform pattern matching in searches without regard to case; see the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.2, Regular Expression General Requirements .

−n *number*	Specify the number of lines per screenful. The *number* argument is a positive decimal integer. The **−n** option shall override any values obtained from any other source.
−p *command*	Each time a screen from a new file is displayed or redisplayed (including as a result of *more* commands; for example, **:p**), execute the *more* command(s) in the command arguments in the order specified, as if entered by the user after the first screen has been displayed. No intermediate results shall be displayed (that is, if the command is a movement to a screen different from the normal first screen, only the screen resulting from the command shall be displayed.) If any of the commands fail for any reason, an informational message to this effect shall be written, and no further commands specified using the **−p** option shall be executed for this file.
−s	Behave as if consecutive empty lines were a single empty line.
−t *tagstring*	Write the screenful of the file containing the tag named by the *tagstring* argument. See the *ctags* utility. The tags feature represented by **−t** *tagstring* and the **:t** command is optional. It shall be provided on any system that also provides a conforming implementation of *ctags*; otherwise, the use of **−t** produces undefined results. The filename resulting from the **−t** option shall be logically added as a prefix to the list of command line files, as if specified by the user. If the tag named by the *tagstring* argument is not found, it shall be an error, and *more* shall take no further action. If the tag specifies a line number, the first line of the display shall contain the beginning of that line. If the tag specifies a pattern, the first line of the display shall contain the beginning of the matching text from the first line of the file that contains that pattern. If the line does not exist in the file or matching text is not found, an informational message to this effect shall be displayed, and *more* shall display the default screen as if **−t** had not been specified. If both the **−t** *tagstring* and **−p** *command* options are given, the **−t** *tagstring* shall be processed first; that is, the file and starting line for the display shall be as specified by **−t**, and then the **−p** *more* command shall be executed. If the line (matching text) specified by the **−t** command does not exist (is not found), no **−p** *more* command shall be executed for this file at any time.
−u	Treat a <backspace> as a printable control character, displayed as an implementation-defined character sequence (see the EXTENDED DESCRIPTION section), suppressing backspacing and the special handling that produces underlined or standout mode text on some terminal types. Also, do not ignore a <carriage-return> at the end of a line.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If no *file* operands are specified, the standard input shall be used. If a *file* is '−', the standard input shall be read at that point in the sequence.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is '−'.

INPUT FILES

The input files being examined shall be text files. If standard output is a terminal, standard error shall be used to read commands from the user. If standard output is a terminal, standard error is not readable, and command input is needed, *more* may attempt to obtain user commands from the controlling terminal (for example, **/dev/tty**); otherwise, *more* shall terminate with an error indicating that it was unable to read user commands. If standard output is not a terminal, no error shall result if standard error cannot be opened for reading.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *more*:

COLUMNS	Override the system-selected horizontal display line size. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables for valid values and results when it is unset or null.
EDITOR	Used by the **v** command to select an editor. See the EXTENDED DESCRIPTION section.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes within regular expressions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
LINES	Override the system-selected vertical screen size, used as the number of lines in a screenful. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables for valid values and results when it is unset or null. The **−n** option shall take precedence over the *LINES* variable for determining the number of lines in a screenful.
MORE	Determine a string containing options described in the OPTIONS section preceded with hyphens and <blank>-separated as on the command line. Any command line options shall be processed after those in the *MORE* variable, as if the command line were: `more $MORE options operands` The *MORE* variable shall take precedence over the *TERM* and *LINES* variables for determining the number of lines in a screenful.

TERM	Determine the name of the terminal type. If this variable is unset or null, an unspecified default terminal type is used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be used to write the contents of the input files.

STDERR

The standard error shall be used for diagnostic messages and user commands (see the INPUT FILES section), and, if standard output is a terminal device, to write a prompting string. The prompting string shall appear on the screen line below the last line of the file displayed in the current screenful. The prompt shall contain the name of the file currently being examined and shall contain an end-of-file indication and the name of the next file, if any, when prompting at the end-of-file. If an error or informational message is displayed, it is unspecified whether it is contained in the prompt. If it is not contained in the prompt, it shall be displayed and then the user shall be prompted for a continuation character, at which point another message or the user prompt may be displayed. The prompt is otherwise unspecified. It is unspecified whether informational messages are written for other user commands.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The following subsection describes the behavior of *more* when the standard output is a terminal device. If the standard output is not a terminal device, no options other than **–s** shall have any effect, and all input files shall be copied to standard output otherwise unmodified, at which time *more* shall exit without further action.

The number of lines available per screen shall be determined by the **–n** option, if present, or by examining values in the environment (see the ENVIRONMENT VARIABLES section). If neither method yields a number, an unspecified number of lines shall be used.

The maximum number of lines written shall be one less than this number, because the screen line after the last line written shall be used to write a user prompt and user input. If the number of lines in the screen is less than two, the results are undefined. It is unspecified whether user input is permitted to be longer than the remainder of the single line where the prompt has been written.

The number of columns available per line shall be determined by examining values in the environment (see the ENVIRONMENT VARIABLES section), with a default value as described in Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

Lines that are longer than the display shall be folded; the length at which folding occurs is unspecified, but should be appropriate for the output device. Folding may occur between glyphs of single characters that take up multiple display columns.

When standard output is a terminal and **–u** is not specified, *more* shall treat <backspace>s and <carriage-return>s specially:

- A character, followed first by a sequence of *n* <backspace>s (where *n* is the same as the number of column positions that the character occupies), then by *n* underscore characters ('_'), shall cause that character to be written as underlined text, if the terminal type supports that. The *n* underscore characters, followed first by *n* <backspace>s, then any character with *n* column positions, shall also cause that character to be written as underlined text, if the terminal type supports that.

- A sequence of *n* <backspace>s (where *n* is the same as the number of column positions that the previous character occupies) that appears between two identical printable characters shall cause the first of those two characters to be written as emboldened text (that is, visually brighter, standout mode, or inverse-video mode), if the terminal type supports that, and the second to be discarded. Immediately subsequent occurrences of <backspace>/character pairs for that same character also shall be discarded. (For example, the sequence `"a\ba\ba\ba"` is interpreted as a single emboldened `'a'`.)

- The *more* utility shall logically discard all other <backspace>s from the line as well as the character which precedes them, if any.

- A <carriage-return> at the end of a line shall be ignored, rather than being written as a non-printable character, as described in the next paragraph.

It is implementation-defined how other non-printable characters are written. Implementations should use the same format that they use for the *ex* **print** command; see the OPTIONS section within the *ed* utility. It is unspecified whether a multi-column character shall be separated if it crosses a display line boundary; it shall not be discarded. The behavior is unspecified if the number of columns on the display is less than the number of columns any single character in the line being displayed would occupy.

When each new file is displayed (or redisplayed), *more* shall write the first screen of the file. Once the initial screen has been written, *more* shall prompt for a user command. If the execution of the user command results in a screen that has lines in common with the current screen, and the device has sufficient terminal capabilities, *more* shall scroll the screen; otherwise, it is unspecified whether the screen is scrolled or redrawn.

For all files but the last (including standard input if no file was specified, and for the last file as well, if the −**e** option was not specified), when *more* has written the last line in the file, *more* shall prompt for a user command. This prompt shall contain the name of the next file as well as an indication that *more* has reached end-of-file. If the user command is **f**, <control>-F, <space>, **j**, <newline>, **d**, <control>-D, or **s**, *more* shall display the next file. Otherwise, if displaying the last file, *more* shall exit. Otherwise, *more* shall execute the user command specified.

Several of the commands described in this section display a previous screen from the input stream. In the case that text is being taken from a non-rewindable stream, such as a pipe, it is implementation-defined how much backwards motion is supported. If a command cannot be executed because of a limitation on backwards motion, an error message to this effect shall be displayed, the current screen shall not change, and the user shall be prompted for another command.

If a command cannot be performed because there are insufficient lines to display, *more* shall alert the terminal. If a command cannot be performed because there are insufficient lines to display or a **/** command fails: if the input is the standard input, the last screen in the file may be displayed; otherwise, the current file and screen shall not change, and the user shall be prompted for another command.

The interactive commands in the following sections shall be supported. Some commands can be preceded by a decimal integer, called *count* in the following descriptions. If not specified with the command, *count* shall default to 1. In the following descriptions, *pattern* is a basic regular expression, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions. The term ‘‘examine’’ is historical usage meaning ‘‘open the file for viewing’’; for example, *more* **foo** would be expressed as examining file **foo**.

In the following descriptions, unless otherwise specified, *line* is a line in the *more* display, not a line from the file being examined.

In the following descriptions, the *current position* refers to two things:

1. The position of the current line on the screen
2. The line number (in the file) of the current line on the screen

Usually, the line on the screen corresponding to the current position is the third line on the screen. If this is not possible (there are fewer than three lines to display or this is the first page of the file, or it is the last page of the file), then the current position is either the first or last line on the screen as described later.

Help

Synopsis: `h`

Write a summary of these commands and other implementation-defined commands. The behavior shall be as if the *more* utility were executed with the −**e** option on a file that contained the summary information. The user shall be prompted as described earlier in this section when end-of-file is reached. If the user command is one of those specified to continue to the next file, *more* shall return to the file and screen state from which the **h** command was executed.

Scroll Forward One Screenful

Synopsis:

```
[count]f
[count]<control>-F
```

Scroll forward *count* lines, with a default of one screenful. If *count* is more than the screen size, only the final screenful shall be written.

Scroll Backward One Screenful

Synopsis:

```
[count]b
[count]<control>-B
```

Scroll backward *count* lines, with a default of one screenful (see the −**n** option). If *count* is more than the screen size, only the final screenful shall be written.

Scroll Forward One Line

Synopsis:

```
[count]<space>
[count]j
[count]<newline>
```

Scroll forward *count* lines. The default *count* for the <space> shall be one screenful; for **j** and <newline>, one line. The entire *count* lines shall be written, even if *count* is more than the screen size.

Scroll Backward One Line

Synopsis: `[count]k`

Scroll backward *count* lines. The entire *count* lines shall be written, even if *count* is more than the screen size.

Scroll Forward One Half Screenful

Synopsis:
```
[count]d
[count]<control>-D
```

Scroll forward *count* lines, with a default of one half of the screen size. If *count* is specified, it shall become the new default for subsequent **d**, <control>-D, and **u** commands.

Skip Forward One Line

Synopsis:
```
[count]s
```

Display the screenful beginning with the line *count* lines after the last line on the current screen. If *count* would cause the current position to be such that less than one screenful would be written, the last screenful in the file shall be written.

Scroll Backward One Half Screenful

Synopsis:
```
[count]u
[count]<control>-U
```

Scroll backward *count* lines, with a default of one half of the screen size. If *count* is specified, it shall become the new default for subsequent **d**, <control>–D, **u**, and <control>–U commands. The entire *count* lines shall be written, even if *count* is more than the screen size.

Go to Beginning of File

Synopsis:
```
[count]g
```

Display the screenful beginning with line *count*.

Go to End-of-File

Synopsis:
```
[count]G
```

If *count* is specified, display the screenful beginning with the line *count*. Otherwise, display the last screenful of the file.

Refresh the Screen

Synopsis:
```
r
<control>-L
```

Refresh the screen.

Discard and Refresh

Synopsis:
```
R
```

Refresh the screen, discarding any buffered input. If the current file is non-seekable, buffered input shall not be discarded and the **R** command shall be equivalent to the **r** command.

Mark Position

Synopsis: `m`*`letter`*

Mark the current position with the letter named by *letter*, where *letter* represents the name of one of the lowercase letters of the portable character set. When a new file is examined, all marks may be lost.

Return to Mark

Synopsis: `'`*`letter`*

Return to the position that was previously marked with the letter named by *letter*, making that line the current position.

Return to Previous Position

Synopsis: `''`

Return to the position from which the last large movement command was executed (where a ‘‘large movement’’ is defined as any movement of more than a screenful of lines). If no such movements have been made, return to the beginning of the file.

Search Forward for Pattern

Synopsis: **[***`count`***]**`/`**[**`!`**]***`pattern`*`<newline>`

Display the screenful beginning with the *count*th line containing the pattern. The search shall start after the first line currently displayed. The null regular expression (`'/'` followed by a <newline>) shall repeat the search using the previous regular expression, with a default *count*. If the character `'!'` is included, the matching lines shall be those that do not contain the *pattern*. If no match is found for the *pattern*, a message to that effect shall be displayed.

Search Backward for Pattern

Synopsis: **[***`count`***]**`?`**[**`!`**]***`pattern`*`<newline>`

Display the screenful beginning with the *count*th previous line containing the pattern. The search shall start on the last line before the first line currently displayed. The null regular expression (`'?'` followed by a <newline>) shall repeat the search using the previous regular expression, with a default *count*. If the character `'!'` is included, matching lines shall be those that do not contain the *pattern*.

If no match is found for the *pattern*, a message to that effect shall be displayed.

Repeat Search

Synopsis: **[***`count`***]**`n`

Repeat the previous search for *count*th line containing the last *pattern* (or not containing the last *pattern*, if the previous search was `"/!"` or `"?!"`).

Repeat Search in Reverse

Synopsis: `[count]N`

Repeat the search in the opposite direction of the previous search for the *count*th line containing the last *pattern* (or not containing the last *pattern*, if the previous search was `"/!"` or `"?!"`).

Examine New File

Synopsis: `:e [filename]<newline>`

Examine a new file. If the *filename* argument is not specified, the current file (see the **:n** and **:p** commands below) shall be re-examined. The *filename* shall be subjected to the process of shell word expansions (see Section 2.6 (on page 2238)); if more than a single pathname results, the effects are unspecified. If *filename* is a number sign (`'#'`), the previously examined file shall be re-examined. If *filename* is not accessible for any reason (including that it is a non-seekable file), an error message to this effect shall be displayed and the current file and screen shall not change.

Examine Next File

Synopsis: `[count]:n`

Examine the next file. If a number *count* is specified, the *count*th next file shall be examined. If *filename* refers to a non-seekable file, the results are unspecified.

Examine Previous File

Synopsis: `[count]:p`

Examine the previous file. If a number *count* is specified, the *count*th previous file shall be examined. If *filename* refers to a non-seekable file, the results are unspecified.

Go to Tag

Synopsis: `:t tagstring<newline>`

If the file containing the tag named by the *tagstring* argument is not the current file, examine the file, as if the **:e** command was executed with that file as the argument. Otherwise, or in addition, display the screenful beginning with the tag, as described for the **–t** option (see the OPTIONS section). If the *ctags* utility is not supported by the system, the use of **:t** produces undefined results.

Invoke Editor

Synopsis: `v`

Invoke an editor to edit the current file being examined. If standard input is being examined, the results are unspecified. The name of the editor shall be taken from the environment variable *EDITOR*, or shall default to *vi*. If the last pathname component in *EDITOR* is either *vi* or *ex*, the editor shall be invoked with a **–c** *linenumber* command line argument, where *linenumber* is the line number of the file line containing the display line currently displayed as the first line of the screen. It is implementation-defined whether line-setting options are passed to editors other than *vi* and *ex*.

When the editor exits, *more* shall resume with the same file and screen as when the editor was invoked.

Display Position

Synopsis: `=`
`<control>-G`

Write a message for which the information references the first byte of the line after the last line of the file on the screen. This message shall include the name of the file currently being examined, its number relative to the total number of files there are to examine, the line number in the file, the byte number and the total bytes in the file, and what percentage of the file precedes the current position. If *more* is reading from standard input, or the file is shorter than a single screen, the line number, the byte number, the total bytes, and the percentage need not be written.

Quit

Synopsis: `q`
`:q`
`ZZ`

Exit *more*.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If an error is encountered accessing a file when using the **:n** command, *more* shall attempt to examine the next file in the argument list, but the final exit status shall be affected. If an error is encountered accessing a file via the **:p** command, *more* shall attempt to examine the previous file in the argument list, but the final exit status shall be affected. If an error is encountered accessing a file via the **:e** command, *more* shall remain in the current file and the final exit status shall not be affected.

APPLICATION USAGE

When the standard output is not a terminal, only the −**s** filter-modification option is effective. This is based on historical practice. For example, a typical implementation of *man* pipes its output through *more* −**s** to squeeze excess white space for terminal users. When *man* is piped to *lp*, however, it is undesirable for this squeezing to happen.

EXAMPLES

The −**p** allows arbitrary commands to be executed at the start of each file. Examples are:

more −**p G** *file1 file2*
Examine each file starting with its last screenful.

more −**p** 100 *file1 file2*
Examine each file starting with line 100 in the current position (usually the third line, so line 98 would be the first line written).

more −**p** /100 *file1 file2*
Examine each file starting with the first line containing the string `"100"` in the current position

RATIONALE

The *more* utility, available in BSD and BSD-derived systems, was chosen as the prototype for the POSIX file display program since it is more widely available than either the public-domain program *less* or than *pg*, a pager provided in System V. The 4.4 BSD *more* is the model for the features selected; it is almost fully upward-compatible from the 4.3 BSD version in wide use and has become more amenable for *vi* users. Several features originally derived from various file editors, found in both *less* and *pg*, have been added to this volume of IEEE Std 1003.1-200x as they have proved extremely popular with users.

There are inconsistencies between *more* and *vi* that result from historical practice. For example, the single-character commands **h**, **f**, **b**, and <space> are screen movers in *more*, but cursor movers in *vi*. These inconsistencies were maintained because the cursor movements are not applicable to *more* and the powerful functionality achieved without the use of the control key justifies the differences.

The tags interface has been included in a program that is not a text editor because it promotes another degree of consistent operation with *vi*. It is conceivable that the paging environment of *more* would be superior for browsing source code files in some circumstances.

The operating mode referred to for block-mode terminals effectively adds a <newline> to each Synopsis line that currently has none. So, for example, **d**<newline> would page one screenful. The mode could be triggered by a command line option, environment variable, or some other method. The details are not imposed by this volume of IEEE Std 1003.1-200x because there are so few systems known to support such terminals. Nevertheless, it was considered that all systems should be able to support *more* given the exception cited for this small community of terminals because, in comparison to *vi*, the cursor movements are few and the command set relatively amenable to the optional <newline>s.

Some versions of *more* provide a shell escaping mechanism similar to the *ex* **!** command. The standard developers did not consider that this was necessary in a paginator, particularly given the wide acceptance of multiple window terminals and job control features. (They chose to retain such features in the editors and *mailx* because the shell interaction also gives an opportunity to modify the editing buffer, which is not applicable to *more*).

The −**p** (position) option replaces the **+** command because of the Utility Syntax Guidelines. In early proposals, it took a *pattern* argument, but historical *less* provided the *more* general facility of a command. It would have been desirable to use the same −**c** as *ex* and *vi*, but the letter was already in use.

The text stating ‘‘from a non-rewindable stream ... implementations may limit the amount of backwards motion supported’’ would allow an implementation that permitted no backwards motion beyond text already on the screen. It was not possible to require a minimum amount of backwards motion that would be effective for all conceivable device types. The implementation should allow the user to back up as far as possible, within device and reasonable memory allocation constraints.

Historically, non-printable characters were displayed using the ARPA standard mappings, which are as follows:

1. Printable characters are left alone.
2. Control characters less than \177 are represented as followed by the character offset from the `'@'` character in the ASCII map; for example, \007 is represented as `'G'`.
3. \177 is represented as followed by `'?'`.

The display of characters having their eighth bit set was less standard. Existing implementations use hex (0x00), octal (\000), and a meta-bit display. (The latter displayed characters with their eighth bit set as the two characters `"M−,"` followed by the seven bit display as described previously.) The latter probably has the best claim to historical practice because it was used with the −**v** option of 4 BSD and 4 BSD-derived versions of the *cat* utility since 1980.

No specific display format is required by IEEE Std 1003.1-200x. Implementations are encouraged to conform to historic practice in the absence of any strong reason to diverge.

FUTURE DIRECTIONS

None.

SEE ALSO

ctags, *ed*, *ex*, *vi*

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The obsolescent SYNOPSIS is removed.

The utility has been extensively reworked for alignment with the IEEE P1003.2b draft standard:

- Changes have been made as result of IEEE PASC Interpretations 1003.2 #37 and #109.
- The *more* utility should be able to handle underlined and emboldened displays of characters that are wider than a single column position.

NAME

mv — move files

SYNOPSIS

```
mv [−fi] source_file target_file

mv [−fi] source_file... target_file
```

DESCRIPTION

In the first synopsis form, the *mv* utility shall move the file named by the *source_file* operand to the *destination* specified by the *target_file*. This first synopsis form is assumed when the final operand does not name an existing directory and is not a symbolic link referring to an existing directory.

In the second synopsis form, *mv* shall move each file named by a *source_file* operand to a *destination* file in the existing directory named by the *target_dir* operand, or referenced if *target_dir* is a symbolic link referring to an existing directory. The *destination* path for each *source_file* shall be the concatenation of the target directory, a single slash character, and the last pathname component of the *source_file*. This second form is assumed when the final operand names an existing directory.

If any operand specifies an existing file of a type not specified by the System Interfaces volume of IEEE Std 1003.1-200x, the behavior is implementation-defined.

For each *source_file* the following steps shall be taken:

1. If the destination path exists, the **−f** option is not specified, and either of the following conditions is true:

    a. The permissions of the destination path do not permit writing and the standard input is a terminal.

    b. The **−i** option is specified.

    the *mv* utility shall write a prompt to standard error and read a line from standard input. If the response is not affirmative, *mv* shall do nothing more with the current *source_file* and go on to any remaining *source_file*s.

2. The *mv* utility shall perform actions equivalent to the *rename*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called with the following arguments:

    a. The *source_file* operand is used as the *old* argument.

    b. The destination path is used as the *new* argument.

    If this succeeds, *mv* shall do nothing more with the current *source_file* and go on to any remaining *source_file*s. If this fails for any reasons other than those described for the *errno* [EXDEV] in the System Interfaces volume of IEEE Std 1003.1-200x, *mv* shall write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

3. If the destination path exists, and it is a file of type directory and *source_file* is not a file of type directory, or it is a file not of type directory and *source_file* is a file of type directory, *mv* shall write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

4. If the destination path exists, *mv* shall attempt to remove it. If this fails for any reason, *mv* shall write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

5. The file hierarchy rooted in *source_file* shall be duplicated as a file hierarchy rooted in the *destination* path. If *source_file* or any of the files below it in the hierarchy are symbolic links, the links themselves shall be duplicated, including their contents, rather than any files to which they refer. The following characteristics of each file in the file hierarchy shall be duplicated:

   - The time of last data modification and time of last access
   - The user ID and group ID
   - The file mode

   If the user ID, group ID, or file mode of a regular file cannot be duplicated, the file mode bits S_ISUID and S_ISGID shall not be duplicated.

   When files are duplicated to another file system, the implementation may require that the process invoking *mv* has read access to each file being duplicated.

   If the duplication of the file hierarchy fails for any reason, *mv* shall write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

   If the duplication of the file characteristics fails for any reason, *mv* shall write a diagnostic message to standard error, but this failure shall not cause *mv* to modify its exit status.

6. The file hierarchy rooted in *source_file* shall be removed. If this fails for any reason, *mv* shall write a diagnostic message to the standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

OPTIONS

The *mv* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

–f	Do not prompt for confirmation if the destination path exists. Any previous occurrences of the **–i** option is ignored.
–i	Prompt for confirmation if the destination path exists. Any previous occurrences of the **–f** option is ignored.

Specifying more than one of the **–f** or **–i** options shall not be considered an error. The last option specified shall determine the behavior of *mv*.

OPERANDS

The following operands shall be supported:

source_file	A pathname of a file or directory to be moved.
target_file	A new pathname for the file or directory being moved.
target_dir	A pathname of an existing directory into which to move the input files.

STDIN

The standard input shall be used to read an input line in response to each prompt specified in the STDERR section. Otherwise, the standard input shall not be used. |

INPUT FILES

The input files specified by each *source_file* operand can be of any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *mv*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the behavior of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_MESSAGES	Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Prompts shall be written to the standard error under the conditions specified in the DESCRIPTION section. The prompts shall contain the *destination* pathname, but their format is otherwise unspecified. Otherwise, the standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output files may be of any file type.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All input files were moved successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If the copying or removal of *source_file* is prematurely terminated by a signal or error, *mv* may leave a partial copy of *source_file* at the source or destination. The *mv* utility shall not modify both *source_file* and the destination path simultaneously; termination at any point shall leave either *source_file* or the destination path complete.

APPLICATION USAGE

Some implementations mark for update the *st_ctime* field of renamed files and some do not. Applications which make use of the *st_ctime* field may behave differently with respect to renamed files unless they are designed to allow for either behavior.

EXAMPLES

If the current directory contains only files **a** (of any type defined by the System Interfaces volume of IEEE Std 1003.1-200x), **b** (also of any type), and a directory **c**:

```
mv a b c
mv c d
```

results with the original files **a** and **b** residing in the directory **d** in the current directory.

RATIONALE

Early proposals diverged from the SVID and BSD historical practice in that they required that when the destination path exists, the −**f** option is not specified, and input is not a terminal, *mv* fails. This was done for compatibility with *cp*. The current text returns to historical practice. It should be noted that this is consistent with the *rename*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, which does not require write permission on the target.

For absolute clarity, paragraph (1), describing the behavior of *mv* when prompting for confirmation, should be interpreted in the following manner:

```
if (exists AND (NOT f_option) AND
    ((not_writable AND input_is_terminal) OR i_option))
```

The −**i** option exists on BSD systems, giving applications and users a way to avoid accidentally unlinking files when moving others. When the standard input is not a terminal, the 4.3 BSD *mv* deletes all existing destination paths without prompting, even when −**i** is specified; this is inconsistent with the behavior of the 4.3 BSD *cp* utility, which always generates an error when the file is unwritable and the standard input is not a terminal. The standard developers decided that use of −**i** is a request for interaction, so when the *destination* path exists, the utility takes instructions from whatever responds to standard input.

The *rename*( ) function is able to move directories within the same file system. Some historical versions of *mv* have been able to move directories, but not to a different file system. The standard developers considered that this was an annoying inconsistency, so this volume of IEEE Std 1003.1-200x requires directories to be able to be moved even across file systems. There is no −**R** option to confirm that moving a directory is actually intended, since such an option was not required for moving directories in historical practice. Requiring the application to specify it sometimes, depending on the destination, seemed just as inconsistent. The semantics of the *rename*( ) function were preserved as much as possible. For example, *mv* is not permitted to ‘‘rename’’ files to or from directories, even though they might be empty and removable.

Historic implementations of *mv* did not exit with a non-zero exit status if they were unable to duplicate any file characteristics when moving a file across file systems, nor did they write a diagnostic message for the user. The former behavior has been preserved to prevent scripts from breaking; a diagnostic message is now required, however, so that users are alerted that the file characteristics have changed.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application not using the −**f** option or using the −**i** option relies on the system to provide the most suitable dialog directly with the user, based on the behavior specified.

When *mv* is dealing with a single file system and *source_file* is a symbolic link, the link itself is moved as a consequence of the dependence on the *rename*( ) functionality, per the DESCRIPTION. Across file systems, this has to be made explicit.

FUTURE DIRECTIONS

None.

SEE ALSO

cp, *ln*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The *mv* utility is changed to describe processing of symbolic links as specified in the IEEE P1003.2b draft standard.

The APPLICATION USAGE section is added.

NAME

newgrp — change to a new group

SYNOPSIS

UP `newgrp [−l][group`

DESCRIPTION

The *newgrp* utility shall create a new shell execution environment with a new real and effective group identification. Of the attributes listed in Section 2.12 (on page 2263), the new shell execution environment shall retain the working directory, file creation mask, and exported variables from the previous environment (that is, open files, traps, unexported variables, alias definitions, shell functions, and *set* options may be lost). All other aspects of the process environment that are preserved by the *exec* family of functions defined in the System Interfaces volume of IEEE Std 1003.1-200x shall also be preserved by *newgrp*; whether other aspects are preserved is unspecified.

A failure to assign the new group identifications (for example, for security or password-related reasons) shall not prevent the new shell execution environment from being created.

The *newgrp* utility shall affect the supplemental groups for the process as follows:

- On systems where the effective group ID is normally in the supplementary group list (or whenever the old effective group ID actually is in the supplementary group list):
  - If the new effective group ID is also in the supplementary group list, *newgrp* shall change the effective group ID.
  - If the new effective group ID is not in the supplementary group list, *newgrp* shall add the new effective group ID to the list, if there is room to add it.
- On systems where the effective group ID is not normally in the supplementary group list (or whenever the old effective group ID is not in the supplementary group list):
  - If the new effective group ID is in the supplementary group list, *newgrp* shall delete it.
  - If the old effective group ID is not in the supplementary list, *newgrp* shall add it if there is room.

Note: The System Interfaces volume of IEEE Std 1003.1-200x does not specify whether the effective group ID of a process is included in its supplementary group list.

With no operands, *newgrp* shall change the effective group back to the groups identified in the user's user entry, and shall set the list of supplementary groups to that set in the user's group database entries.

If a password is required for the specified group, and the user is not listed as a member of that group in the group database, the user shall be prompted to enter the correct password for that group. If the user is listed as a member of that group, no password shall be requested. If no password is required for the specified group, it is implementation-defined whether users not listed as members of that group can change to that group. Whether or not a password is required, implementation-defined system accounting or security mechanisms may impose additional authorization restrictions that may cause *newgrp* to write a diagnostic message and suppress the changing of the group identification. |

OPTIONS

The *newgrp* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

–l	(The letter ell.) Change the environment to what would be expected if the user actually logged in again.

OPERANDS

The following operand shall be supported:

group	A group name from the group database or a non-negative numeric group ID. Specifies the group ID to which the real and effective group IDs shall be set. If *group* is a non-negative numeric string and exists in the group database as a group name (see *getgrnam*( )), the numeric group ID associated with that group name shall be used as the group ID.

STDIN

Not used.

INPUT FILES

The file **/dev/tty** shall be used to read a single line of text for password checking, when one is required.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *newgrp*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used for diagnostic messages and a prompt string for a password, if one is required. Diagnostic messages may be written in cases where the exit status is not available. See the EXIT STATUS section.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If *newgrp* succeeds in creating a new shell execution environment, whether or not the group identification was changed successfully, the exit status shall be the exit status of the shell. Otherwise, the following exit value shall be returned:

>0 An error occurred.

CONSEQUENCES OF ERRORS

The invoking shell may terminate.

APPLICATION USAGE

There is no convenient way to enter a password into the Group Database. Use of group passwords is not encouraged, because by their very nature they encourage poor security practices. Group passwords may disappear in the future.

A common implementation of *newgrp* is that the current shell uses *exec* to overlay itself with *newgrp*, which in turn overlays itself with a new shell after changing group. On some implementations, however, this may not occur and *newgrp* may be invoked as a subprocess.

The *newgrp* command is intended only for use from an interactive terminal. It does not offer a useful interface for the support of applications.

The exit status of *newgrp* is generally inapplicable. If *newgrp* is used in a script, in most cases it successfully invokes a new shell and the rest of the original shell script is bypassed when the new shell exits. Used interactively, *newgrp* displays diagnostic messages to indicate problems. But usage such as:

```
newgrp foo
echo $?
```

is not useful because the new shell might not have access to any status *newgrp* may have generated (and most historical systems do not provide this status). A zero status echoed here does not necessarily indicate that the user has changed to the new group successfully. Following *newgrp* with the *id* command provides a portable means of determining whether the group change was successful or not.

EXAMPLES

None.

RATIONALE

Most historical implementations use one of the *exec* functions to implement the behavior of *newgrp*. Errors detected before the *exec* leave the environment unchanged, while errors detected after the *exec* leave the user in a changed environment. While it would be useful to have *newgrp* issue a diagnostic message to tell the user that the environment changed, it would be inappropriate to require this change to some historical implementations.

The password mechanism is allowed in the group database, but how this would be implemented is not specified.

The *newgrp* utility was retained in this volume of IEEE Std 1003.1-200x, even given the existence of the multiple group permissions feature in the System Interfaces volume of IEEE Std 1003.1-200x, for several reasons. First, in some implementations, the group ownership of a newly created file is determined by the group of the directory in which the file is created, as allowed by the System Interfaces volume of IEEE Std 1003.1-200x; on other implementations, the group ownership of a newly created file is determined by the effective group ID. On implementations of the latter type, *newgrp* allows files to be created with a specific group ownership. Finally, many implementations use the real group ID in accounting, and on such systems, *newgrp* allows the accounting identity of the user to be changed.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the System Interfaces volume of IEEE Std 1003.1-200x, *exec*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The obsolescent SYNOPSIS is removed.

The text describing supplemental groups is no longer conditional on {NGROUPS_MAX} being greater than 1. This is because {NGROUPS_MAX} now has a minimum value of 8. This is a FIPS requirement.

NAME

nice — invoke a utility with an altered nice value

SYNOPSIS

UP
```
nice [−n increment] utility [argument...]
```

DESCRIPTION

The *nice* utility shall invoke a utility, requesting that it be run with a different nice value (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.239, Nice Value). With no options and only if the user has appropriate privileges, the executed utility shall be run with a nice value that is some implementation-defined quantity less than or equal to the nice value of the current process. If the user lacks appropriate privileges to affect the nice value in the requested manner, the *nice* utility shall not affect the nice value; in this case, a warning message may be written to standard error, but this shall not prevent the invocation of *utility* or affect the exit status.

OPTIONS

The *nice* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option is supported:

−n *increment*	A positive or negative decimal integer which shall have the same effect on the execution of the utility as if the utility had called the *nice*( ) function with the numeric value of the *increment* option-argument. |

OPERANDS

The following operands shall be supported:

utility	The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 (on page 2266), the results are undefined.
argument	Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *nice*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
	PATH	Determine the search path used to locate the utility to be invoked. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If the *utility* utility is invoked, the exit status of *nice* shall be the exit status of *utility*; otherwise, the *nice* utility shall exit with one of the following values:

1-125	An error occurred in the *nice* utility.
126	The utility specified by *utility* was found but could not be invoked.
127	The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The only guaranteed portable uses of this utility are:

nice utility
    Run *utility* with the default lower nice value.

nice –**n** <*positive integer*> *utility*
    Run *utility* with a lower nice value.

On some implementations they have no discernible effect on the invoked utility and on some others they are exactly equivalent.

Historical systems have frequently supported the <*positive integer*> up to 20. Since there is no error penalty associated with guessing a number that is too high, users without access to the system conformance document (to see what limits are actually in place) could use the historical 1 to 20 range or attempt to use very large numbers if the job should be truly low priority.

The nice value value of a process can be displayed using the command:

```
ps –o nice
```

The *command*, *env*, *nice*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

None.

RATIONALE

Due to the text about the limits of the nice value being implementation-defined, *nice* is not actually required to change the nice value of the executed command; the limits could be zero differences from the system default, although the implementor is required to document this fact in the conformance document.

The 4.3 BSD version of *nice* does not check if *increment* is a valid decimal integer. The command *nice* −**x** *utility*, for example, would be treated the same as the command *nice* −−**1** *utility*. If the user does not have appropriate privileges, this results in a ‘‘permission denied’’ error. This is considered a bug.

When a user without appropriate privileges gives a negative *increment*, System V treats it like the command *nice* −**0** *utility*, while 4.3 BSD writes a ‘‘permission denied’’ message and does not run the utility. Neither was considered clearly superior, so the behavior was left unspecified.

The C shell has a built-in version of *nice* that has a different interface from the one described in this volume of IEEE Std 1003.1-200x.

The term ‘‘utility’’ is used, rather than ‘‘command’’, to highlight the fact that shell compound commands, pipelines, and so on, cannot be used. Special built-ins also cannot be used. However, ‘‘utility’’ includes user application programs and shell scripts, not just utilities defined in this volume of IEEE Std 1003.1-200x.

Historical implementations of *nice* provide a nice value range of 40 or 41 discrete steps, with the default nice value being the midpoint of that range. By default, they lower the nice value of the executed utility by 10.

Some historical documentation states that the *increment* value must be within a fixed range. This is misleading; the valid *increment* values on any invocation are determined by the current process nice value, which is not always the default.

The definition of nice value is not intended to suggest that all processes in a system have priorities that are comparable. Scheduling policy extensions such as the realtime priorities in the System Interfaces volume of IEEE Std 1003.1-200x make the notion of a single underlying priority for all scheduling policies problematic. Some implementations may implement the *nice*-related features to affect all processes on the system, others to affect just the general time-sharing activities implied by this volume of IEEE Std 1003.1-200x, and others may have no effect at all. Because of the use of ‘‘implementation-defined’’ in *nice* and *renice*, a wide range of implementation strategies are possible. |

FUTURE DIRECTIONS

None.

SEE ALSO

renice, the System Interfaces volume of IEEE Std 1003.1-200x, *nice*( ) |

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The obsolescent SYNOPSIS is removed.

NAME

nl — line numbering filter

SYNOPSIS

XSI
```
nl [−p][−b type][−d delim][−f type][−h type][−i incr][−l num][−n format]
    [−s sep][−v startnum][−w width][file]
```

DESCRIPTION

The *nl* utility shall read lines from the named *file* or the standard input if no *file* is named and shall reproduce the lines to standard output. Lines shall be numbered on the left. Additional functionality may be provided in accordance with the command options in effect.

The *nl* utility views the text it reads in terms of logical pages. Line numbering shall be reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (for example, no numbering of header and footer lines while numbering blank lines only in the body).

The starts of logical page sections shall be signaled by input lines containing nothing but the following delimiter characters:

Line	Start of
`\:\:\:`	Header
`\:\:`	Body
`\:`	Footer

Unless otherwise specified, *nl* shall assume the text being read is in a single logical page body.

OPTIONS

The *nl* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines. Only one file can be named.

The following options shall be supported:

−b *type* Specify which logical page body lines shall be numbered. Recognized *types* and their meaning are:

- **a** Number all lines.
- **t** Number only non-empty lines.
- **n** No line numbering.
- **p***string* Number only lines that contain the basic regular expression specified in *string*.

The default *type* for logical page body shall be **t** (text lines numbered).

−d *delim* Specify the delimiter characters that indicate the start of a logical page section. These can be changed from the default characters `"\:"` to two user-specified characters. If only one character is entered, the second character shall remain the default character `':'`.

−f *type* Specify the same as **b** *type* except for footer. The default for logical page footer shall be **n** (no lines numbered).

−h *type* Specify the same as **b** *type* except for header. The default *type* for logical page header shall be **n** (no lines numbered).

−i *incr*	Specify the increment value used to number logical page lines. The default shall be 1.
−l *num*	Specify the number of blank lines to be considered as one. For example, **−l 2** results in only the second adjacent blank line being numbered (if the appropriate **−h a**, **−b a**, or **−f a** option is set). The default shall be 1.
−n *format*	Specify the line numbering format. Recognized values are: **ln**, left justified, leading zeros suppressed; **rn**, right justified, leading zeros suppressed; **rz**, right justified, leading zeros kept. The default *format* shall be **rn** (right justified).
−p	Specify that numbering should not be restarted at logical page delimiters.
−s *sep*	Specify the characters used in separating the line number and the corresponding text line. The default *sep* shall be a <tab>.
−v *startnum*	Specify the initial value used to number logical page lines. The default shall be 1.
−w *width*	Specify the number of characters to be used for the line number. The default *width* shall be 6.

OPERANDS

The following operand shall be supported:

file	A pathname of a text file to be line-numbered.

STDIN

The standard input is a text file that is used if no *file* operand is given.

INPUT FILES

The input file named by the *file* operand is a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *nl*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the behavior of character classes within regular expressions, and for deciding which characters are in character class **graph** (for the **−b t**, **−f t**, and **−h t** options).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be a text file in the following format:

```
"%s%s%s", <line number>, <separator>, <input line>
```

where <*line number*> is one of the following numeric formats:

`%6d`	When the **rn** format is used (the default; see −**n**).
`%06d`	When the **rz** format is used.
`%−6d`	When the *ln* format is used.
<empty>	When line numbers are suppressed for a portion of the page; the <*separator*> is also suppressed.

In the preceding list, the number 6 is the default width; the −**w** option can change this value.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

In using the −**d** *delim* option, care should be taken to escape characters that have special meaning to the command interpreter.

EXAMPLES

The command:

```
nl −v 10 −i 10 −d \!+ file1
```

numbers *file1* starting at line number 10 with an increment of 10. The logical page delimiter is `"!+"`. Note that the `'!'` has to be escaped when using *csh* as a command interpreter because of its history substitution syntax. For *ksh* and *sh* the escape is not necessary, but does not do any harm.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pr

CHANGE HISTORY

First released in Issue 2.

Issue 5

The option [–**f** *type*] is added to the SYNOPSIS. The option descriptions are presented in alphabetic order. The description of –**bt** is changed to ''Number only non-empty lines''.

Issue 6

The obsolescent behavior allowing the options to be intermingled with the optional *file* operand is removed.

NAME

nm — write the name list of an object file (**DEVELOPMENT**)

SYNOPSIS

UP SD XSI
```
nm [−APv][−efox][ −g| −u][−t format] file...
```

DESCRIPTION

This utility shall be provided on systems that support both the User Portability Utilities option and the Software Development Utilities option. On other systems it is optional. Certain options are only available on XSI-conformant systems.

The *nm* utility shall display symbolic information appearing in the object file, executable file or object-file library named by *file*. If no symbolic information is available for a valid input file, the *nm* utility shall report that fact, but not consider it an error condition.

XSI The default base used when numeric values are written is unspecified. On XSI-conformant systems, it shall be decimal.

OPTIONS

The *nm* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

	−A	Write the full pathname or library name of an object on each line.
XSI	**−e**	Write only external (global) and static symbol information.
XSI	**−f**	Produce full output. Write redundant symbols (**.text**, **.data**, and **.bss**), normally suppressed.
	−g	Write only external (global) symbol information.
XSI	**−o**	Write numeric values in octal (equivalent to **−t o**).
	−P	Write information in a portable output format, as specified in the STDOUT section.
	−t *format*	Write each numeric value in the specified format. The format shall be dependent on the single character used as the *format* option-argument:
XSI		`d` The offset is written in decimal (default).
		`o` The offset is written in octal.
		`x` The offset is written in hexadecimal.
	−u	Write only undefined symbols.
	−v	Sort output by value instead of alphabetically.
XSI	**−x**	Write numeric values in hexadecimal (equivalent to **−t x**).

OPERANDS

The following operand shall be supported:

file	A pathname of an object file, executable file, or object-file library.

STDIN

See the INPUT FILES section.

INPUT FILES

The input file shall be an object file, an object-file library whose format is the same as those produced by the *ar* utility for link editing, or an executable file. The *nm* utility may accept additional implementation-defined object library formats for the input file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *nm*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for character collation information for the symbol-name and symbol-value collation sequences.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If symbolic information is present in the input files, then for each file or for each member of an archive, the *nm* utility shall write the following information to standard output. By default, the format is unspecified, but the output shall be sorted alphabetically by symbol name:

- Library or object name, if **–A** is specified
- Symbol name
- Symbol type, which shall either be one of the following single characters or an implementation-defined type represented by a single character:

`A`	Global absolute symbol.
`a`	Local absolute symbol.
`B`	Global ‘‘bss’’ (that is, uninitialized data space) symbol.
`b`	Local bss symbol.
`D`	Global data symbol.
`d`	Local data symbol.
`T`	Global text symbol.
`t`	Local text symbol.
`U`	Undefined symbol.

- Value of the symbol
- The size associated with the symbol, if applicable

This information may be supplemented by additional information specific to the implementation.

If the −**P** option is specified, the previous information shall be displayed using the following portable format. The three versions differ depending on whether −**t d**, −**t o**, or −**t x** was specified, respectively:

```
"%s%s %s %d %d\n", <library/object name>, <name>, <type>,
    <value>, <size>
```

```
"%s%s %s %o %o\n", <library/object name>, <name>, <type>,
    <value>, <size>
```

```
"%s%s %s %x %x\n", <library/object name>, <name>, <type>,
    <value>, <size>
```

where

<*library/object name*> shall be formatted as follows:

- If −**A** is not specified, <*library/object name*> shall be an empty string.
- If −**A** is specified and the corresponding *file* operand does not name a library:

  ```
  "%s: ", <file>
  ```

- If −**A** is specified and the corresponding *file* operand names a library. In this case, <*object file*> shall name the object file in the library containing the symbol being described:

  ```
  "%s[%s]: ", <file>, <object file>
  ```

If −**A** is not specified, then if more than one *file* operand is specified or if only one *file* operand is specified and it names a library, *nm* shall write a line identifying the object containing the following symbols before the lines containing those symbols, in the form:

- If the corresponding *file* operand does not name a library:

  ```
  "%s:\n", <file>
  ```

- If the corresponding *file* operand names a library; in this case, <*object file*> shall be the name of the file in the library containing the following symbols:

  ```
  "%s[%s]:\n", <file>, <object file>
  ```

If −**P** is specified, but −**t** is not, the format shall be as if −**t x** had been specified.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Mechanisms for dynamic linking make this utility less meaningful when applied to an executable file because a dynamically linked executable may omit numerous library routines that would be found in a statically linked executable.

EXAMPLES

None.

RATIONALE

Historical implementations of *nm* have used different bases for numeric output and supplied different default types of symbols that were reported. The −**t** *format* option, similar to that used in *od* and *strings*, can be used to specify the numeric base; −**g** and −**u** can be used to restrict the amount of output or the types of symbols included in the output.

The compromise of using −**t** *format versus* using −**d**, −**o**, and other similar options was necessary because of differences in the meaning of −**o** between implementations. The −**o** option from BSD has been provided here as −**A** to avoid confusion with the −**o** from System V (which has been provided here as −**t** and as −**o** on XSI-conformant systems).

The option list was significantly reduced from that provided by historical implementations.

The *nm* description is a subset of both the System V and BSD *nm* utilities with no specified default output.

It was recognized that mechanisms for dynamic linking make this utility less meaningful when applied to an executable file (because a dynamically linked executable file may omit numerous library routines that would be found in a statically linked executable file), but the value of *nm* during software development was judged to outweigh other limitations.

The default output format of *nm* is not specified because of differences in historical implementations. The −**P** option was added to allow some type of portable output format. After a comparison of the different formats used in SunOS, BSD, SVR3, and SVR4, it was decided to create one that did not match the current format of any of these four systems. The format devised is easy to parse by humans, easy to parse in shell scripts, and does not need to vary depending on locale (because no English descriptions are included). All of the systems currently have the information available to use this format.

The format given in *nm* STDOUT uses spaces between the fields, which may be any number of <blank>s required to align the columns. The single-character types were selected to match historical practice, and the requirement that implementation additions also be single characters made parsing the information easier for shell scripts.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *c99*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as supported when both the User Portability Utilities option and the Software Development Utilities option are supported.

NAME

nohup — invoke a utility immune to hangups

SYNOPSIS

```
nohup utility [argument...]
```

DESCRIPTION

The *nohup* utility shall invoke the utility named by the *utility* operand with arguments supplied as the *argument* operands. At the time the named *utility* is invoked, the SIGHUP signal shall be set to be ignored.

If the standard output is a terminal, all output written by the named *utility* to its standard output shall be appended to the end of the file **nohup.out** in the current directory. If **nohup.out** cannot be created or opened for appending, the output shall be appended to the end of the file **nohup.out** in the directory specified by the *HOME* environment variable. If neither file can be created or opened for appending, *utility* shall not be invoked. If a file is created, the file's permission bits shall be set to S_IRUSR | S_IWUSR.

If the standard error is a terminal, all output written by the named *utility* to its standard error shall be redirected to the same file descriptor as the standard output.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

utility	The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 (on page 2266), the results are undefined.
argument	Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *nohup*:

HOME	Determine the pathname of the user's home directory: if the output file **nohup.out** cannot be created in the current directory, the *nohup* utility shall use the directory named by *HOME* to create the file.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of

diagnostic messages written to standard error.

XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
PATH	Determine the search path that is used to locate the utility to be invoked. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

The *nohup* utility shall take the standard action for all signals except that SIGHUP shall be ignored.

STDOUT

If the standard output is not a terminal, the standard output of *nohup* shall be the standard output generated by the execution of the *utility* specified by the operands. Otherwise, nothing shall be written to the standard output.

STDERR

If the standard output is a terminal, a message shall be written to the standard error, indicating the name of the file to which the output is being appended. The name of the file shall be either **nohup.out** or **$HOME/nohup.out**.

OUTPUT FILES

If the standard output is a terminal, all output written by the named *utility* to the standard output and standard error is appended to the file **nohup.out**, which is created if it does not already exist.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

126	The utility specified by *utility* was found but could not be invoked.
127	An error occurred in the *nohup* utility or the utility specified by *utility* could not be found.

Otherwise, the exit status of *nohup* shall be that of the utility specified by the *utility* operand.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *command*, *env*, *nice*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

It is frequently desirable to apply *nohup* to pipelines or lists of commands. This can be done by placing pipelines and command lists in a single file; this file can then be invoked as a utility, and the *nohup* applies to everything in the file.

Alternatively, the following command can be used to apply *nohup* to a complex command:

```
nohup sh −c 'complex-command-line'
```

RATIONALE

The 4.3 BSD version ignores SIGTERM and SIGHUP, and if **./nohup.out** cannot be used, it fails instead of trying to use **$HOME/nohup.out**.

The *csh* utility has a built-in version of *nohup* that acts differently from the POSIX Shell and Utilities *nohup*.

The term *utility* is used, rather than *command*, to highlight the fact that shell compound commands, pipelines, special built-ins, and so on, cannot be used directly. However, *utility* includes user application programs and shell scripts, not just the standard utilities.

Historical versions of the *nohup* utility use default file creation semantics. Some more recent versions use the permissions specified here as an added security precaution.

Some historical implementations ignore SIGQUIT in addition to SIGHUP; others ignore SIGTERM. An early proposal allowed, but did not require, SIGQUIT to be ignored. Several reviewers objected that *nohup* should only modify the handling of SIGHUP as required by this volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the System Interfaces volume of IEEE Std 1003.1-200x, *signal*( )

CHANGE HISTORY

First released in Issue 2.

NAME

od — dump files in various formats

SYNOPSIS

```
od [−v][−A address_base][−j skip][−N count][−t type_string]...
    [file...]
```

XSI
```
od [−bcdosx][file] [[+]offset[.][b]]
```

DESCRIPTION

The *od* utility shall write the contents of its input files to standard output in a user-specified format.

OPTIONS

XSI The *od* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the order of presentation of the **−t** options and the **−bcdosx** options is significant.

The following options shall be supported:

−A *address_base*
Specify the input offset base. See the EXTENDED DESCRIPTION section. The application shall ensure that the *address_base* option-argument is a character. The characters `'d'`, `'o'`, and `'x'` specify that the offset base shall be written in decimal, octal, or hexadecimal, respectively. The character `'n'` specifies that the offset shall not be written.

XSI **−b** Interpret bytes in octal. This shall be equivalent to **−t o1**.

XSI **−c** Interpret bytes as characters specified by the current setting of the *LC_CTYPE* category. Certain non-graphic characters appear as C escapes: `"NUL=\0"`, `"BS=\b"`, `"FF=\f"`, `"NL=\n"`, `"CR=\r"`, `"HT=\t"`; others appear as 3-digit octal numbers.

XSI **−d** Interpret *words* (two-byte units) in unsigned decimal. This shall be equivalent to **−t u2**.

−j *skip* Jump over *skip* bytes from the beginning of the input. The *od* utility shall read or seek past the first *skip* bytes in the concatenated input files. If the combined input is not at least *skip* bytes long, the *od* utility shall write a diagnostic message to standard error and exit with a non-zero exit status.

By default, the *skip* option-argument shall be interpreted as a decimal number. With a leading 0x or 0X, the offset shall be interpreted as a hexadecimal number; otherwise, with a leading `'0'`, the offset shall be interpreted as an octal number. Appending the character `'b'`, `'k'`, or `'m'` to offset shall cause it to be interpreted as a multiple of 512, 1 024, or 1 048 576 bytes, respectively. If the *skip* number is hexadecimal, any appended `'b'` shall be considered to be the final hexadecimal digit.

−N *count* Format no more than *count* bytes of input. By default, *count* shall be interpreted as a decimal number. With a leading 0x or 0X, *count* shall be interpreted as a hexadecimal number; otherwise, with a leading `'0'`, it shall be interpreted as an octal number. If *count* bytes of input (after successfully skipping, if **−j** *skip* is specified) are not available, it shall not be considered an error; the *od* utility shall format the input that is available.

XSI **−o** Interpret *words* (two-byte units) in octal. This shall be equivalent to **−t o2**.

XSI **−s** Interpret *words* (two-byte units) in signed decimal. This shall be equivalent to **−t d2**.

−t *type_string*

Specify one or more output types. See the EXTENDED DESCRIPTION section. The application shall ensure that the *type_string* option-argument is a string specifying the types to be used when writing the input data. The string shall consist of the type specification characters `a`, `c`, `d`, `f`, `o`, `u`, and `x`, specifying named character, character, signed decimal, floating point, octal, unsigned decimal, and hexadecimal, respectively. The type specification characters `d`, `f`, `o`, `u`, and `x` can be followed by an optional unsigned decimal integer that specifies the number of bytes to be transformed by each instance of the output type. The type specification character `f` can be followed by an optional `F`, `D`, or `L` indicating that the conversion should be applied to an item of type **float**, **double**, or **long double**, respectively. The type specification characters `d`, `o`, `u` and `x` can be followed by an optional `C`, `S`, `I`, or `L` indicating that the conversion should be applied to an item of type **char**, **short**, **int**, or **long**, respectively. Multiple types can be concatenated within the same *type_string* and multiple **−t** options can be specified. Output lines shall be written for each type specified in the order in which the type specification characters are specified.

−v Write all input data. Without the **−v** option, any number of groups of output lines, which would be identical to the immediately preceding group of output lines (except for the byte offsets), shall be replaced with a line containing only an asterisk (`'*'`).

XSI **−x** Interpret *words* (two-byte units) in hexadecimal. This shall be equivalent to **−t x2**.

XSI Multiple types can be specified by using multiple **−bcdostx** options. Output lines are written for each type specified in the order in which the types are specified.

OPERANDS

The following operands shall be supported:

file A pathname of a file to be read. If no *file* operands are specified, the standard input shall be used.

If there are no more than two operands, none of the **−A**, **−j**, **−N**, or **−t** options is specified, and either of the following is true: the first character of the last operand is a plus sign (`'+'`), or there are two operands and the first character of the last operand is numeric; XSI the last operand shall be interpreted as an offset operand on XSI-conformant systems. Under these conditions, the results are unspecified on systems that are not XSI-conformant systems.

XSI **[+]***offset***[.][b]** The *offset* operand specifies the offset in the file where dumping is to commence. This operand is normally interpreted as octal bytes. If `'.'` is appended, the offset shall be interpreted in decimal. If `'b'` is appended, the offset shall be interpreted in units of 512 bytes.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *od*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_NUMERIC
Determine the locale for selecting the radix character used when writing floating-point formatted output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the EXTENDED DESCRIPTION section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The *od* utility shall copy sequentially each input file to standard output, transforming the input XSI data according to the output types specified by the −**t** options or the −**bcdosx** options. If no output type is specified, the default output shall be as if −**t oS** had been specified.

The number of bytes transformed by the output type specifier `c` may be variable depending on the *LC_CTYPE* category.

The default number of bytes transformed by output type specifiers `d`, `f`, `o`, `u`, and `x` corresponds to the various C-language types as follows. If the *c99* compiler is present on the system, these specifiers shall correspond to the sizes used by default in that compiler. Otherwise, these sizes may vary among systems that conform to IEEE Std 1003.1-200x.

- For the type specifier characters `d`, `o`, `u`, and `x`, the default number of bytes shall correspond to the size of the underlying implementation's basic integer type. For these specifier characters, the implementation shall support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **char**, **short**, **int**, and **long**. These numbers can also be specified by an application as the characters `'C'`, `'S'`, `'I'`, and `'L'`, respectively. The implementation shall also support the values 1, 2, 4, and 8, even if it provides no C-Language types of those sizes. The implementation shall support the decimal value corresponding to the C-language type **long long**. The byte order used when interpreting numeric values is implementation-defined, but shall correspond to the order in which a constant of the corresponding type is stored in memory on the system.

- For the type specifier character `f`, the default number of bytes shall correspond to the number of bytes in the underlying implementation's basic double precision floating-point data type. The implementation shall support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **float, double**, and **long double**. These numbers can also be specified by an application as the characters `'F'`, `'D'`, and `'L'`, respectively.

The type specifier character `a` specifies that bytes shall be interpreted as named characters from the International Reference Version (IRV) of the ISO/IEC 646:1991 standard. Only the least significant seven bits of each byte shall be used for this type specification. Bytes with the values listed in the following table shall be written using the corresponding names for those characters.

Table 4-12 Named Characters in *od*

Value	Name	Value	Name	Value	Name	Value	Name
\000	**nul**	\001	**soh**	\002	**stx**	\003	**etx**
\004	**eot**	\005	**enq**	\006	**ack**	\007	**bel**
\010	**bs**	\011	**ht**	\012	**lf** or **nl***	\013	**vt**
\014	**ff**	\015	**cr**	\016	**so**	\017	**si**
\020	**dle**	\021	**dc1**	\022	**dc2**	\023	**dc3**
\024	**dc4**	\025	**nak**	\026	**syn**	\027	**etb**
\030	**can**	\031	**em**	\032	**sub**	\033	**esc**
\034	**fs**	\035	**gs**	\036	**rs**	\037	**us**
\040	**sp**	\177	**del**				

Note: The `"\012"` value may be written either as **lf** or **nl**.

The type specifier character `c` specifies that bytes shall be interpreted as characters specified by the current setting of the *LC_CTYPE* locale category. Characters listed in the table in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`) shall be written as the corresponding escape sequences, except that backslash shall be written as a single backslash and a NUL shall be written as `'\0'`. Other non-printable characters shall be written as one three-digit octal number for each byte in the character. If the size of a byte on the system is greater than nine bits, the format used for non-printable characters is implementation-defined. Printable multi-byte characters shall be written in the area corresponding to the first byte of the character; the two-character sequence `"**"` shall be written in the area corresponding to each remaining byte in the character, as an indication that the character is continued. When either the **−j** *skip* or **−N** *count* option is specified along with the `c` type specifier, and this results in an attempt to start or finish in the middle of a multi-byte character, the result is implementation-defined.

The input data shall be manipulated in blocks, where a block is defined as a multiple of the least common multiple of the number of bytes transformed by the specified output types. If the least common multiple is greater than 16, the results are unspecified. Each input block shall be written as transformed by each output type, one per written line, in the order that the output types were specified. If the input block size is larger than the number of bytes transformed by the output type, the output type shall sequentially transform the parts of the input block, and the output from each of the transformations shall be separated by one or more <blank>s.

If, as a result of the specification of the –**N** option or end-of-file being reached on the last input file, input data only partially satisfies an output type, the input shall be extended sufficiently with null bytes to write the last byte of the input.

Unless –**A n** is specified, the first output line produced for each input block shall be preceded by the input offset, cumulative across input files, of the next byte to be written. The format of the input offset is unspecified; however, it shall not contain any <blank>s, shall start at the first character of the output line, and shall be followed by one or more <blank>s. In addition, the offset of the byte following the last byte written shall be written after all the input data has been processed, but shall not be followed by any <blank>s.

If no –**A** option is specified, the input offset base is unspecified.

EXIT STATUS

The following exit values shall be returned:

0	All input files were processed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

XSI-conformant applications are warned not to use filenames starting with `'+'` or a first operand starting with a numeric character so that the old functionality can be maintained by implementations, unless they specify one of the –**A**, –**j**, or –**N** options. To guarantee that one of these filenames is always interpreted as a filename, an application could always specify the address base format with the –**A** option.

EXAMPLES

If a file containing 128 bytes with decimal values zero to 127, in increasing order, is supplied as standard input to the command:

```
od –A d –t a
```

on an implementation using an input block size of 16 bytes, the standard output, independent of the current locale setting, would be similar to:

```
0000000 nul soh stx etx eot enq ack bel  bs  ht  nl  vt  ff  cr  so  si
0000016 dle dc1 dc2 dc3 dc4 nak syn etb can  em sub esc  fs  gs  rs  us
0000032  sp   !   "   #   $   %   &   ’   (   )   *   +   ,   –   .   /
0000048   0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
0000064   @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
0000080   P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _
0000096   ‘   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
0000112   p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~ del
0000128
```

Note that this volume of IEEE Std 1003.1-200x allows **nl** or **lf** to be used as the name for the ISO/IEC 646:1991 standard IRV character with decimal value 10. The IRV names this character **lf** (line feed), but traditional implementations have referred to this character as newline (**nl**) and the POSIX locale character set symbolic name for the corresponding character is a <newline>.

The command:

```
od –A o –t o2x2x –n 18
```

on a system with 32-bit words and an implementation using an input block size of 16 bytes could write 18 bytes in approximately the following format:

```
0000000 032056 031440 041123 042040 052516 044530 020043 031464
            342e    3320    4253    4420    554e    4958    2023    3334
                342e3320        42534420        554e4958        20233334
0000020 032472
            353a
                353a0000
0000022
```

The command:

```
od −A d −t f −t o4 −t x4 −n 24 −j 0x15
```

on a system with 64-bit doubles (for example, IEEE Std 754-1985 double precision floating-point format) would skip 21 bytes of input data and then write 24 bytes in approximately the following format:

```
0000000    1.00000000000000e+00    1.57350000000000e+01
        07774000000 00000000000 10013674121 35341217270
            3ff00000    00000000    402f3851    eb851eb8
0000016    1.40668230000000e+02
        10030312542 04370303230
            40619562    23e18698
0000024
```

RATIONALE

The *od* utility went through several names in early proposals, including *hd*, *xd*, and most recently *hexdump*. There were several objections to all of these based on the following reasons:

- The *hd* and *xd* names conflicted with historical utilities that behaved differently.
- The *hexdump* description was much more complex than needed for a simple dump utility.
- The *od* utility has been available on all historical implementations and there was no need to create a new name for a utility so similar to the historical *od* utility.

The original reasons for not standardizing historical *od* were also fairly widespread. Those reasons are given below along with rationale explaining why the standard developers believe that this version does not suffer from the indicated problem:

- The BSD and System V versions of *od* have diverged, and the intersection of features provided by both does not meet the needs of the user community. In fact, the System V version only provides a mechanism for dumping octal bytes and **short**s, signed and unsigned decimal **short**s, hexadecimal **short**s, and ASCII characters. BSD added the ability to dump **float**s, **double**s, named ASCII characters, and octal, signed decimal, unsigned decimal, and hexadecimal **long**s. The version presented here provides more normalized forms for dumping bytes, **short**s, **int**s, and **long**s in octal, signed decimal, unsigned decimal, and hexadecimal; **float**, **double**, and **long double**; and named ASCII as well as current locale characters.
- It would not be possible to come up with a compatible superset of the BSD and System V flags that met the requirements of the standard developers. The historical default *od* output is the specified default output of this utility. None of the option letters chosen for this version of *od* conflict with any of the options to historical versions of *od*.
- On systems with different sizes for **short**, **int**, and **long**, there was no way to ask for dumps of **int**s, even in the BSD version. Because of the way options are named, the name space could not be extended to solve these problems. This is why the **−t** option was added (with type specifiers more closely matched to the *printf*( ) formats used in the rest of this volume of

IEEE Std 1003.1-200x) and the optional field sizes were added to the `d`, `f`, `o`, `u`, and `x` type specifiers. It is also one of the reasons why the historical practice was not mandated as a required obsolescent form of *od*. (Although the old versions of *od* are not listed as an obsolescent form, implementations are urged to continue to recognize the older forms for several more years.) The `a`, `c`, `f`, `o`, and `x` types match the meaning of the corresponding format characters in the historical implementations of *od* except for the default sizes of the fields converted. The `d` format is signed in this volume of IEEE Std 1003.1-200x to match the *printf*( ) notation. (Historical versions of *od* used `d` as a synonym for `u` in this version. The System V implementation uses `s` for signed decimal; BSD uses `i` for signed decimal and `s` for null-terminated strings.) Other than `d` and `u`, all of the type specifiers match format characters in the historical BSD version of **od**.

The sizes of the C-language types **char**, **short**, **int**, **long**, **float**, **double**, and **long double** are used even though it is recognized that there may be zero or more than one compiler for the C language on an implementation and that they may use different sizes for some of these types. (For example, one compiler might use 2 bytes **short**s, 2 bytes **int**s, and 4 bytes **long**s, while another compiler (or an option to the same compiler) uses 2 bytes **short**s, 4 bytes **int**s, and 4 bytes **long**s.) Nonetheless, there has to be a basic size known by the implementation for these types, corresponding to the values reported by invocations of the *getconf* utility when called with *system_var* operands {UCHAR_MAX}, {USHORT_MAX}, {UINT_MAX}, and {ULONG_MAX} for the types **char**, **short**, **int**, and **long**, respectively. There are similar constants required by the ISO C standard, but not required by the System Interfaces volume of IEEE Std 1003.1-200x or this volume of IEEE Std 1003.1-200x. They are {FLT_MANT_DIG}, {DBL_MANT_DIG}, and {LDBL_MANT_DIG} for the types **float**, **double**, and **long double**, respectively. If the optional *c99* utility is provided by the implementation and used as specified by this volume of IEEE Std 1003.1-200x, these are the sizes that would be provided. If an option is used that specifies different sizes for these types, there is no guarantee that the *od* utility is able to interpret binary data output by such a program correctly.

This volume of IEEE Std 1003.1-200x requires that the numeric values of these lengths be recognized by the *od* utility and that symbolic forms also be recognized. Thus, a conforming application can always look at an array of **unsigned long** data elements using *od* −**t** *uL*.

- The method of specifying the format for the address field based on specifying a starting offset in a file unnecessarily tied the two together. The −**A** option now specifies the address base and the −**S** option specifies a starting offset.
- It would be difficult to break the dependence on U.S. ASCII to achieve an internationalized utility. It does not seem to be any harder for *od* to dump characters in the current locale than it is for the *ed* or *sed* **l** commands. The **c** type specifier does this without difficulty and is completely compatible with the historical implementations of the **c** format character when the current locale uses a superset of the ISO/IEC 646:1991 standard as a codeset. The **a** type specifier (from the BSD **a** format character) was left as a portable means to dump ASCII (or more correctly ISO/IEC 646:1991 standard (IRV)) so that headers produced by *pax* could be deciphered even on systems that do not use the ISO/IEC 646:1991 standard as a subset of their base codeset.

The use of `"**"` as an indication of continuation of a multi-byte character in **c** specifier output was chosen based on seeing an implementation that uses this method. The continuation bytes have to be marked in a way that is not ambiguous with another single-byte or multi-byte character.

An early proposal used −**S** and −**n**, respectively, for the −**j** and −**N** options eventually selected. These were changed to avoid conflicts with historical implementations.

The original standard specified **−t o2** as the default when no output type was given. This was changed to **−t oS** (the length of a **short**) to accommodate a supercomputer implementation that historically used 64 bits as its default (and that defined shorts as 64 bits). This change should not affect conforming applications. The requirement to support lengths of 1, 2, and 4 was added at the same time to address an historical implementation that had no two-byte data types in its C compiler.

The use of a basic integer data type is intended to allow the implementation to choose a word size commonly used by applications on that architecture.

FUTURE DIRECTIONS

All option and operand interfaces marked as extensions may be withdrawn in a future issue.

SEE ALSO

c99, *sed*

CHANGE HISTORY

First released in Issue 2.

Issue 5

In the description of the **−c** option, the phrase ''This is equivalent to **−t c**.'' is deleted.

The FUTURE DIRECTIONS section has been modified.

Issue 6

The *od* utility is changed to remove the assumption that **short** was a two-byte entity, as per the revisions in the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ''must'' for application requirements.

NAME

paste — merge corresponding or subsequent lines of files

SYNOPSIS

```
paste [−s][−d list] file...
```

DESCRIPTION

The *paste* utility shall concatenate the corresponding lines of the given input files, and writes the resulting lines to standard output.

The default operation of *paste* shall concatenate the corresponding lines of the input files. The <newline> of every line except the line from the last input file shall be replaced with a <tab>.

If an end-of-file condition is detected on one or more input files, but not all input files, *paste* shall behave as though empty lines were read from the files on which end-of-file was detected, unless the −**s** option is specified.

OPTIONS

The *paste* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−**d** *list* Unless a backslash character appears in *list*, each character in *list* is an element specifying a delimiter character. If a backslash character appears in *list*, the backslash character and one or more characters following it are an element specifying a delimiter character as described below. These elements specify one or more delimiters to use, instead of the default <tab>, to replace the <newline> of the input lines. The elements in *list* shall be used circularly; that is, when the list is exhausted the first element from the list is reused. When the −**s** option is specified:

- The last <newline> in a file shall not be modified.
- The delimiter shall be reset to the first element of list after each *file* operand is processed.

When the −**s** option is not specified:

- The <newline>s in the file specified by the last *file* operand shall not be modified.
- The delimiter shall be reset to the first element of list each time a line is processed from each file.

If a backslash character appears in *list*, it and the character following it shall be used to represent the following delimiter characters:

`\n` <newline>.

`\t` <tab>.

`\\` Backslash character.

`\0` Empty string (not a null character). If `'\0'` is immediately followed by the character `'x'`, the character `'X'`, or any character defined by the *LC_CTYPE* **digit** keyword (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale), the results are unspecified.

If any other characters follow the backslash, the results are unspecified.

−**s** Concatenate all of the lines of each separate input file in command line order. The <newline> of every line except the last line in each input file shall be replaced with the <tab>, unless otherwise specified by the **−d** option.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If ′−′ is specified for one or more of the *file*s, the standard input shall be used; the standard input shall be read one line at a time, circularly, for each instance of ′−′. Implementations shall support pasting of at least 12 *file* operands.

STDIN

The standard input shall be used only if one or more *file* operands is ′−′. See the INPUT FILES section.

INPUT FILES

The input files shall be text files, except that line lengths shall be unlimited.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *paste*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Concatenated lines of input files shall be separated by the <tab> (or other characters under the control of the **−d** option) and terminated by a <newline>.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If one or more input files cannot be opened when the **-s** option is not specified, a diagnostic message shall be written to standard error, but no output is written to standard output. If the **-s** option is specified, the *paste* utility shall provide the default behavior described in Section 1.11 (on page 2221).

APPLICATION USAGE

When the escape sequences of the *list* option-argument are used in a shell script, they must be quoted; otherwise, the shell treats the `'\\'` as a special character.

Conforming applications should only use the specific backslash escaped delimiters presented in this volume of IEEE Std 1003.1-200x. Historical implementations treat `'\x'`, where `'x'` is not in this list, as `'x'`, but future implementations are free to expand this list to recognize other common escapes similar to those accepted by *printf* and other standard utilities.

Most of the standard utilities work on text files. The *cut* utility can be used to turn files with arbitrary line lengths into a set of text files containing the same data. The *paste* utility can be used to create (or recreate) files with arbitrary line lengths. For example, if *file* contains long lines:

```
cut -b 1-500 -n file > file1
cut -b 501- -n file > file2
```

creates **file1** (a text file) with lines no longer than 500 bytes (plus the <newline>) and **file2** that contains the remainder of the data from *file*. Note that **file2** is not a text file if there are lines in *file* that are longer than 500 + {LINE_MAX} bytes. The original file can be recreated from **file1** and **file2** using the command:

```
paste -d "\0" file1 file2 > file
```

The commands:

```
paste -d "\0" ...
paste -d "" ...
```

are not necessarily equivalent; the latter is not specified by this volume of IEEE Std 1003.1-200x and may result in an error. The construct `'\0'` is used to mean ‘‘no separator’’ because historical versions of *paste* did not follow the syntax guidelines, and the command:

```
paste -d"" ...
```

could not be handled properly by *getopt*( ).

EXAMPLES

1. Write out a directory in four columns:

   ```
   ls | paste - - - -
   ```

2. Combine pairs of lines from a file into single lines:

   ```
   paste -s -d "\t\n" file
   ```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cut, *grep*, *pr*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

patch — apply changes to files

SYNOPSIS

UP

```
patch [−blNR][ −c| −e| −n][−d dir][−D define][−i patchfile]
    [−o outfile][−p num][−r rejectfile][file]
```

DESCRIPTION

The *patch* utility shall read a source (patch) file containing any of the three forms of difference (diff) listings produced by the *diff* utility (normal, context or in the style of *ed*) and apply those differences to a file. By default, *patch* shall read from the standard input.

The *patch* utility shall attempt to determine the type of the *diff* listing, unless overruled by a **−c**, **−e**, or **−n** option.

If the patch file contains more than one patch, *patch* shall attempt to apply each of them as if they came from separate patch files. (In this case, the application shall ensure that the name of the patch file is determinable for each *diff* listing.)

OPTIONS

The *patch* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−b Save a copy of the original contents of each modified file, before the differences are applied, in a file of the same name with the suffix **.orig** appended to it. If the file already exists, it shall be overwritten; if multiple patches are applied to the same file, the **.orig** file shall be written only for the first patch. When the **−o** *outfile* option is also specified, *file***.orig** shall not be created but, if *outfile* already exists, *outfile***.orig** shall be created.

−c Interpret the patch file as a context difference (the output of the utility *diff* when the **−c** or **−C** options are specified).

−d *dir* Change the current directory to *dir* before processing as described in the EXTENDED DESCRIPTION section.

−D *define* Mark changes with one of the following C preprocessor constructs:

```
#ifdef define
...
#endif

#ifndef define
...
#endif
```

optionally combined with the C preprocessor construct **#else**.

−e Interpret the patch file as an *ed* script, rather than a *diff* script.

−i *patchfile* Read the patch information from the file named by the pathname *patchfile*, rather than the standard input.

−l (The letter ell.) Cause any sequence of <blank>s in the difference script to match any sequence of <blank>s in the input file. Other characters shall be matched exactly.

−n	Interpret the script as a normal difference.
−N	Ignore patches where the differences have already been applied to the file; by default, already-applied patches shall be rejected.
−o *outfile*	Instead of modifying the files (specified by the *file* operand or the difference listings) directly, write a copy of the file referenced by each patch, with the appropriate differences applied, to *outfile*. Multiple patches for a single file shall be applied to the intermediate versions of the file created by any previous patches, and shall result in multiple, concatenated versions of the file being written to *outfile*.
−p *num*	For all pathnames in the patch file that indicate the names of files to be patched, delete *num* pathname components from the beginning of each pathname. If the pathname in the patch file is absolute, any leading slashes shall be considered the first component (that is, **−p 1** shall remove the leading slashes). Specifying **−p 0** shall cause the full pathname to be used. If **−p** is not specified, only the basename (the final pathname component) shall be used.
−R	Reverse the sense of the patch script; that is, assume that the difference script was created from the new version to the old version. The **−R** option cannot be used with *ed* scripts. The *patch* utility shall attempt to reverse each portion of the script before applying it. Rejected differences shall be saved in swapped format. If this option is not specified, and until a portion of the patch file is successfully applied, *patch* attempts to apply each portion in its reversed sense as well as in its normal sense. If the attempt is successful, the user shall be prompted to determine whether the **−R** option should be set.
−r *rejectfile*	Override the default reject filename. In the default case, the reject file shall have the same name as the output file, with the suffix **.rej** appended to it; see **Patch Application** (on page 2893).

OPERANDS

The following operand shall be supported:

file	A pathname of a file to patch.

STDIN

See the INPUT FILES section.

INPUT FILES

Input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *patch*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

LC_TIME Determine the locale for recognizing the format of file timestamps written by the *diff* utility in a context-difference input file.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used for diagnostic and informational messages.

OUTPUT FILES

The output of the *patch* utility, the save files (**.orig** suffixes) and the reject files (**.rej** suffixes) shall be text files.

EXTENDED DESCRIPTION

A patchfile may contain patching instructions for more than one file; filenames shall be determined as specified in **Filename Determination** (on page 2893). When the **–b** option is specified, for each patched file, the original shall be saved in a file of the same name with the suffix **.orig** appended to it.

For each patched file, a reject file may also be created as noted in **Patch Application** (on page 2893). In the absence of a **–r** option, the name of this file shall be formed by appending the suffix **.rej** to the original filename.

Patchfile Format

The patch file shall contain zero or more lines of header information followed by one or more patches. Each patch shall contain zero or more lines of filename identification in the format produced by *diff* **–c**, and one or more sets of *diff* output, which are customarily called *hunks.*

The *patch* utility shall recognize the following expression in the header information:

Index: *pathname*
The file to be patched is named *pathname*.

If all lines (including headers) within a patch begin with the same leading sequence of <blank>s, the *patch* utility shall remove this sequence before proceeding. Within each patch, if the type of difference is context, the *patch* utility shall recognize the following expressions:

*** *filename timestamp*
The patches arose from *filename*.

– – – *filename timestamp*
The patches should be applied to *filename*.

Each hunk within a patch shall be the *diff* output to change a line range within the original file. The line numbers for successive hunks within a patch shall occur in ascending order.

Filename Determination

If no *file* operand is specified, *patch* shall perform the following steps to determine the filename to use:

1. If the type of *diff* is context, the *patch* utility shall delete pathname components (as specified by the −**p** option) from the filename on the line beginning with `"***"`, then test for the existence of this file relative to the current directory (or the directory specified with the −**d** option). If the file exists, the *patch* utility shall use this filename.
2. If the type of *diff* is context, the *patch* utility shall delete the pathname components (as specified by the −**p** option) from the filename on the line beginning with "−−−", then test for the existence of this file relative to the current directory (or the directory specified with the −**d** option). If the file exists, the *patch* utility shall use this filename.
3. If the header information contains a line beginning with the string **Index:**, the *patch* utility shall delete pathname components (as specified by the −**p** option) from this line, then test for the existence of this file relative to the current directory (or the directory specified with the −**d** option). If the file exists, the *patch* utility shall use this filename.
4. XSI If an **SCCS** directory exists in the current directory, *patch* shall attempt to perform a *get* −**e** **SCCS/s.***filename* command to retrieve an editable version of the file. If the file exists, the *patch* utility shall use this filename.
5. The *patch* utility shall write a prompt to standard output and request a filename interactively from the controlling terminal (for example, **/dev/tty**).

Patch Application

If the −**c**, −**e**, or −**n** option is present, the *patch* utility shall interpret information within each hunk as a context difference, an *ed* difference or a normal difference, respectively. In the absence of any of these options, the *patch* utility shall determine the type of difference based on the format of information within the hunk.

For each hunk, the *patch* utility shall begin to search for the place to apply the patch at the line number at the beginning of the hunk, plus or minus any offset used in applying the previous hunk. If lines matching the hunk context are not found, *patch* shall scan both forwards and backwards at least 1 000 bytes for a set of lines that match the hunk context.

If no such place is found and it is a context difference, then another scan shall take place, ignoring the first and last line of context. If that fails, the first two and last two lines of context shall be ignored and another scan shall be made. Implementations may search more extensively for installation locations.

If no location can be found, the *patch* utility shall append the hunk to the reject file. The rejected hunk shall be written in context-difference format regardless of the format of the patch file. If the input was a normal or *ed–style* difference, the reject file may contain differences with zero lines of context. The line numbers on the hunks in the reject file may be different from the line numbers in the patch file since they shall reflect the approximate locations for the failed hunks in the new file rather than the old one.

If the type of patch is an *ed* diff, the implementation may accomplish the patching by invoking the *ed* utility.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

1 One or more lines were written to a reject file.

>1 An error occurred.

CONSEQUENCES OF ERRORS

Patches that cannot be correctly placed in the file shall be written to a reject file.

APPLICATION USAGE

The −**R** option does not work with *ed* scripts because there is too little information to reconstruct the reverse operation.

The −**p** option makes it possible to customize a patchfile to local user directory structures without manually editing the patchfile. For example, if the filename in the patch file was:

```
/curds/whey/src/blurfl/blurfl.c
```

Setting −**p 0** gives the entire pathname unmodified; −**p 1** gives:

```
curds/whey/src/blurfl/blurfl.c
```

without the leading slash, −**p 4** gives:

```
blurfl/blurfl.c
```

and not specifying −**p** at all gives:

```
blurfl.c .
```

EXAMPLES

None.

RATIONALE

Some of the functionality in historical *patch* implementations was not specified. The following documents those features present in historical implementations that have not been specified.

A deleted piece of functionality was the `'+'` pseudo-option allowing an additional set of options and a patch file operand to be given. This was seen as being insufficiently useful to standardize.

In historical implementations, if the string `"Prereq:"` appeared in the header, the *patch* utility would search for the corresponding version information (the string specified in the header, delimited by <blank>s or the beginning or end of a line or the file) anywhere in the original file. This was deleted as too simplistic and insufficiently trustworthy a mechanism to standardize. For example, if:

```
Prereq: 1.2
```

were in the header, the presence of a delimited 1.2 anywhere in the file would satisfy the prerequisite.

The following options were dropped from historical implementations of *patch* as insufficiently useful to standardize:

−**b** The −**b** option historically provided a method for changing the name extension of the backup file from the default **.orig**. This option has been modified and retained in this volume of IEEE Std 1003.1-200x.

−**F** The −**F** option specified the number of lines of a context diff to ignore when searching for a place to install a patch.

−**f** The −**f** option historically caused *patch* not to request additional information from the user.

−r	The −**r** option historically provided a method of overriding the extension of the reject file from the default **.rej**.
−s	The −**s** option historically caused *patch* to work silently unless an error occurred.
−x	The −**x** option historically set internal debugging flags.

In some file system implementations, the saving of a **.orig** file may produce unwanted results. In the case of 12, 13, or 14-character filenames (on file systems supporting 14-character maximum filenames), the **.orig** file overwrites the new file. The reject file may also exceed this filename limit. It was suggested, due to some historical practice, that a tilde (′˜′) suffix be used instead of **.orig** and some other character instead of the **.rej** suffix. This was rejected because it is not obvious to the user which file is which. The suffixes **.orig** and **.rej** are clearer and more understandable.

The −**b** option has the opposite sense in some historical implementations—do not save the **.orig** file. The default case here is not to save the files, making *patch* behave more consistently with the other standard utilities.

The −**w** option in early proposals was changed to −**l** to match historical practice.

The −**N** option was included because without it, a non-interactive application cannot reject previously applied patches. For example, if a user is piping the output of *diff* into the *patch* utility, and the user only wants to patch a file to a newer version non-interactively, the −**N** option is required.

Changes to the −**l** option description were proposed to allow matching across <newline>s in addition to just <blank>s. Since this is not historical practice, and since some ambiguities could result, it is suggested that future developments in this area utilize another option letter, such as −**L**.

FUTURE DIRECTIONS

None.

SEE ALSO

ed, *diff*

CHANGE HISTORY

First released in Issue 4.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The description of the −**D** option and the steps in **Filename Determination** (on page 2893) are changed to match historical practice as defined in the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

pathchk — check pathnames

SYNOPSIS

`pathchk [−p] pathname...`

DESCRIPTION

The *pathchk* utility shall check that one or more pathnames are valid (that is, they could be used to access or create a file without causing syntax errors) and portable (that is, no filename truncation results). More extensive portability checks are provided by the **−p** option.

By default, the *pathchk* utility shall check each component of each *pathname* operand based on the underlying file system. A diagnostic shall be written for each *pathname* operand that:

- Is longer than {PATH_MAX} bytes (see **Pathname Variable Values** in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers, **<limits.h>**)
- Contains any component longer than {NAME_MAX} bytes in its containing directory
- Contains any component in a directory that is not searchable
- Contains any character in any component that is not valid in its containing directory

The format of the diagnostic message is not specified, but shall indicate the error detected and the corresponding *pathname* operand.

It shall not be considered an error if one or more components of a *pathname* operand do not exist as long as a file matching the pathname specified by the missing components could be created that does not violate any of the checks specified above.

OPTIONS

The *pathchk* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−p Instead of performing checks based on the underlying file system, write a diagnostic for each *pathname* operand that:

- Is longer than {_POSIX_PATH_MAX} bytes (see **Minimum Values** in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers, **<limits.h>**)
- Contains any component longer than {_POSIX_NAME_MAX} bytes
- Contains any character in any component that is not in the portable filename character set

OPERANDS

The following operand shall be supported:

pathname A pathname to be checked.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *pathchk*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All *pathname* operands passed all of the checks.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *test* utility can be used to determine whether a given pathname names an existing file; it does not, however, give any indication of whether or not any component of the pathname was truncated in a directory where the _POSIX_NO_TRUNC feature is not in effect. The *pathchk* utility does not check for file existence; it performs checks to determine whether a pathname does exist or could be created with no pathname component truncation.

The *noclobber* option in the shell (see the *set* (on page 2287) special built-in) can be used to atomically create a file. As with all file creation semantics in the System Interfaces volume of IEEE Std 1003.1-200x, it guarantees atomic creation, but still depends on applications to agree on conventions and cooperate on the use of files after they have been created.

EXAMPLES

To verify that all pathnames in an imported data interchange archive are legitimate and unambiguous on the current system:

```
pax −f archive | sed −e ′/ == .*/s///′ | xargs pathchk
if [ $? −eq 0 ]
then
    pax −r −f archive
```

```
else
    echo Investigate problems before importing files.
    exit 1
fi
```

To verify that all files in the current directory hierarchy could be moved to any system conforming to the System Interfaces volume of IEEE Std 1003.1-200x that also supports the *pax* utility:

```
find . −print | xargs pathchk −p
if [ $? −eq 0 ]
then
    pax −w −f archive .
else
    echo Portable archive cannot be created.
    exit 1
fi
```

To verify that a user-supplied pathname names a readable file and that the application can create a file extending the given path without truncation and without overwriting any existing file:

```
case $− in
    *C*)    reset="";;
    *)      reset="set +C"
            set −C;;
esac
test −r "$path" && pathchk "$path.out" &&
    rm "$path.out" > "$path.out"
if [ $? −ne 0 ]; then
    printf "%s: %s not found or %s.out fails \
creation checks.\n" $0 "$path" "$path"
    $reset    # Reset the noclobber option in case a trap
              # on EXIT depends on it.
    exit 1
fi
$reset
PROCESSING < "$path" > "$path.out"
```

The following assumptions are made in this example:

1. **PROCESSING** represents the code that is used by the application to use **$path** once it is verified that **$path.out** works as intended.

2. The state of the *noclobber* option is unknown when this code is invoked and should be set on exit to the state it was in when this code was invoked. (The **reset** variable is used in this example to restore the initial state.)

3. Note the usage of:

   ```
   rm "$path.out" > "$path.out"
   ```

   a. The *pathchk* command has already verified, at this point, that **$path.out** is not truncated.

   b. With the *noclobber* option set, the shell verifies that **$path.out** does not already exist before invoking *rm*.

c. If the shell succeeded in creating **$path.out**, *rm* removes it so that the application can create the file again in the **PROCESSING** step.

d. If the **PROCESSING** step wants the file to exist already when it is invoked, the:

```
rm "$path.out" > "$path.out"
```

should be replaced with:

```
> "$path.out"
```

which verifies that the file did not already exist, but leaves **$path.out** in place for use by **PROCESSING**.

RATIONALE

The *pathchk* utility is new, commissioned for this volume of IEEE Std 1003.1-200x. It, along with the *set* −**C**(*noclobber*) option added to the shell, replaces the *mktemp*, *validfnam*, and *create* utilities that appeared in early proposals. All of these utilities were attempts to solve several common problems:

- Verify the validity (for several different definitions of ‘‘valid’’) of a pathname supplied by a user, generated by an application, or imported from an external source.
- Atomically create a file.
- Perform various string handling functions to generate a temporary filename.

The *create* utility, included in an early proposal, provided checking and atomic creation in a single invocation of the utility; these are orthogonal issues and need not be grouped into a single utility. Note that the *noclobber* option also provides a way of creating a lock for process synchronization; since it provides an atomic *create*, there is no race between a test for existence and the following creation if it did not exist.

Having a function like *tmpnam*( ) in the ISO C standard is important in many high-level languages. The shell programming language, however, has built-in string manipulation facilities, making it very easy to construct temporary filenames. The names needed obviously depend on the application, but are frequently of a form similar to:

`$TMPDIR/``application_abbreviation`**`$$.`**`suffix`

In cases where there is likely to be contention for a given suffix, a simple shell *for* or *while* loop can be used with the shell *noclobber* option to create a file without risk of collisions, as long as applications trying to use the same filename name space are cooperating on the use of files after they have been created.

FUTURE DIRECTIONS

None.

SEE ALSO

test, Section 2.7 (on page 2244)

CHANGE HISTORY

First released in Issue 4.

NAME

pax — portable archive interchange

SYNOPSIS

```
pax [−cdnv][−H|−L][−f archive][−s replstr]...[pattern...]

pax −r[−cdiknuv][−H|−L][−f archive][−o options]...[−p string]...
    [−s replstr]...[pattern...]

pax −w[−dituvX][−H|−L][−b blocksize][[−a][−f archive][−o options]...
    [−s replstr]...[−x format][file...]

pax −r −w[−diklntuvX][−H|−L][−p string]...[−s replstr]...
    [file...] directory
```

DESCRIPTION

The *pax* utility shall read, write, and write lists of the members of archive files and copy directory hierarchies. A variety of archive formats shall be supported; see the **−x** *format* option.

The action to be taken depends on the presence of the **−r** and **−w** options. The four combinations of **−r** and **−w** are referred to as the four modes of operation: **list**, **read**, **write**, and **copy** modes, corresponding respectively to the four forms shown in the SYNOPSIS section.

list In **list** mode (when neither **−r** nor **−w** are specified), *pax* shall write the names of the members of the archive file read from the standard input, with pathnames matching the specified patterns, to standard output. If a named file is of type directory, the file hierarchy rooted at that file shall be listed as well.

read In **read** mode (when **−r** is specified, but **−w** is not), *pax* shall extract the members of the archive file read from the standard input, with pathnames matching the specified patterns. If an extracted file is of type directory, the file hierarchy rooted at that file shall be extracted as well. The extracted files shall be created performing pathname resolution with the directory in which *pax* was invoked as the current working directory.

If an attempt is made to extract a directory when the directory already exists, this shall not be considered to be an error. If an attempt is made to extract a FIFO when the FIFO already exists, this shall not be considered to be an error.

The ownership, access, and modification times, and file mode of the restored files are discussed under the **−p** option.

write In **write** mode (when **−w** is specified, but **−r** is not), *pax* shall write the contents of the *file* operands to the standard output in an archive format. If no *file* operands are specified, a list of files to copy, one per line, shall be read from the standard input. A file of type directory shall include all of the files in the file hierarchy rooted at the file.

copy In **copy** mode (when both **−r** and **−w** are specified), *pax* shall copy the *file* operands to the destination directory.

If no *file* operands are specified, a list of files to copy, one per line, shall be read from the standard input. A file of type directory shall include all of the files in the file hierarchy rooted at the file.

The effect of the **copy** shall be as if the copied files were written to an archive file and then subsequently extracted, except that there may be hard links between the original and the copied files. If the destination directory is a subdirectory of one of the files to be copied, the results are unspecified. If the destination directory is a file of a type not defined by the System Interfaces volume of IEEE Std 1003.1-200x, the results are implementation-defined; otherwise, it shall be an error for the file named by the *directory* operand not to exist, not be writable by the user, or not be a file of type directory.

In **read** or **copy** modes, if intermediate directories are necessary to extract an archive member, *pax* shall perform actions equivalent to the *mkdir*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x, called with the following arguments:

- The intermediate directory used as the *path* argument
- The value of the bitwise-inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO as the *mode* argument

If any specified *pattern* or *file* operands are not matched by at least one file or archive member, *pax* shall write a diagnostic message to standard error for each one that did not match and exit with a non-zero exit status.

The archive formats described in the EXTENDED DESCRIPTION section shall be automatically detected on input. The default output archive format shall be implementation-defined.

A single archive can span multiple files. The *pax* utility shall determine, in an implementation-defined manner, what file to read or write as the next file.

If the selected archive format supports the specification of linked files, it shall be an error if these files cannot be linked when the archive is extracted. For archive formats that do not store file contents with each name that causes a hard link, if the file that contains the data is not extracted during this *pax* session, either the data shall be restored from the original file, or a diagnostic message shall be displayed with the name of a file that can be used to extract the data. In traversing directories, *pax* shall detect infinite loops; that is, entering a previously visited directory that is an ancestor of the last file visited. When it detects an infinite loop, *pax* shall write a diagnostic message to standard error and shall terminate.

OPTIONS

The *pax* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the order of presentation of the −**o**, −**p**, and −**s** options is significant.

The following options shall be supported:

−**r**	Read an archive file from standard input.
−**w**	Write files to the standard output in the specified archive format.
−**a**	Append files to the end of the archive. It is implementation-defined which devices on the system support appending. Additional file formats unspecified by this volume of IEEE Std 1003.1-200x may impose restrictions on appending.
−**b** *blocksize*	Block the output at a positive decimal integer number of bytes per write to the archive file. Devices and archive formats may impose restrictions on blocking. Blocking shall be automatically determined on input. Conforming applications shall not specify a *blocksize* value larger than 32 256. Default blocking when creating archives depends on the archive format. (See the −**x** option below.)
−**c**	Match all file or archive members except those specified by the *pattern* or *file* operands.
−**d**	Cause files of type directory being copied or archived or archive members of type directory being extracted or listed to match only the file or archive member itself and not the file hierarchy rooted at the file.

−f *archive* Specify the pathname of the input or output archive, overriding the default standard input (in **list** or **read** modes) or standard output (**write** mode).

−H If a symbolic link referencing a file of type directory is specified on the command line, *pax* shall archive the file hierarchy rooted in the file referenced by the link, using the name of the link as the root of the file hierarchy. Otherwise, if a symbolic link referencing a file of any other file type which *pax* can normally archive is specified on the command line, then *pax* shall archive the file referenced by the link, using the name of the link. The default behavior shall be to archive the symbolic link itself.

−i Interactively rename files or archive members. For each archive member matching a *pattern* operand or file matching a *file* operand, a prompt shall be written to the file **/dev/tty**. The prompt shall contain the name of the file or archive member, but the format is otherwise unspecified. A line shall then be read from **/dev/tty**. If this line is blank, the file or archive member shall be skipped. If this line consists of a single period, the file or archive member shall be processed with no modification to its name. Otherwise, its name shall be replaced with the contents of the line. The *pax* utility shall immediately exit with a non-zero exit status if end-of-file is encountered when reading a response or if **/dev/tty** cannot be opened for reading and writing.

The results of extracting a hard link to a file that has been renamed during extraction are unspecified.

−k Prevent the overwriting of existing files.

−l (The letter ell.) In **copy** mode, hard links shall be made between the source and destination file hierarchies whenever possible. If specified in conjunction with **−H** or **−L**, when a symbolic link is encountered, the hard link created in the destination file hierarchy shall be to the file referenced by the symbolic link. If specified when neither **−H** nor **−L** is specified, when a symbolic link is encountered, the implementation shall create a hard link to the symbolic link in the source file hierarchy or copy the symbolic link to the destination.

−L If a symbolic link referencing a file of type directory is specified on the command line or encountered during the traversal of a file hierarchy, *pax* shall archive the file hierarchy rooted in the file referenced by the link, using the name of the link as the root of the file hierarchy. Otherwise, if a symbolic link referencing a file of any other file type which *pax* can normally archive is specified on the command line or encountered during the traversal of a file hierarchy, *pax* shall archive the file referenced by the link, using the name of the link. The default behavior shall be to archive the symbolic link itself.

−n Select the first archive member that matches each *pattern* operand. No more than one archive member shall be matched for each pattern (although members of type directory shall still match the file hierarchy rooted at that file).

−o *options* Provide information to the implementation to modify the algorithm for extracting or writing files. The value of *options* shall consist of one or more comma-separated keywords of the form:

```
keyword[[:]=value][,keyword[[:]=value], ...]
```

Some keywords apply only to certain file formats, as indicated with each description. Use of keywords that are inapplicable to the file format being processed produces undefined results.

Keywords in the *options* argument shall be a string that would be a valid portable filename as described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.276, Portable Filename Character Set.

Note: Keywords are not expected to be filenames, merely to follow the same character composition rules as portable filenames.

Keywords can be preceded with white space. The *value* field shall consist of zero or more characters; within *value*, the application shall precede any literal comma with a backslash, which shall be ignored, but preserves the comma as part of *value*. A comma as the final character, or a comma followed solely by white space as the final characters, in *options* shall be ignored. Multiple −**o** options can be specified; if keywords given to these multiple −**o** options conflict, the keywords and values appearing later in command line sequence shall take precedence and the earlier shall be silently ignored. The following keyword values of *options* shall be supported for the file formats as indicated:

delete=*pattern*
(Applicable only to the −**x** *pax* format.) When used in **write** or **copy** mode, *pax* shall omit from extended header records that it produces any keywords matching the string pattern. When used in **read** or **list** mode, *pax* shall ignore any keywords matching the string pattern in the extended header records. In both cases, matching shall be performed using the pattern matching notation described in Section 2.13.1 (on page 2264) and Section 2.13.2 (on page 2264). For example:

```
−o delete=security.*
```

would suppress security-related information. See **pax Extended Header** (on page 2913) for extended header record keyword usage.

exthdr.name=*string*
(Applicable only to the −**x** *pax* format.) This keyword allows user control over the name that is written into the **ustar** header blocks for the extended header produced under the circumstances described in **pax Header Block** (on page 2912). The name shall be the contents of *string*, after the following character substitutions have been made:

string **Includes:**	**Replaced By:**
`%d`	The directory name of the file, equivalent to the result of the *dirname* utility on the translated pathname.
`%f`	The filename of the file, equivalent to the result of the *basename* utility on the translated pathname.
`%%`	A `'%'` character.

Any other `'%'` characters in *string* produce undefined results.

If no −**o** **exthdr.name=***string* is specified, *pax* shall use the following default value:

```
%d/PaxHeaders/%f
```

globexthdr.name=*string*
(Applicable only to the −**x** *pax* format.) When used in **write** or **copy** mode with the appropriate options, *pax* shall create global extended header records with **ustar** header blocks that will be treated as regular files by previous versions of

pax. This keyword allows user control over the name that is written into the **ustar** header blocks for global extended header records. The name shall be the contents of string, after the following character substitutions have been made:

string **Includes:**	**Replaced By:**
`%n`	An integer that represents the sequence number of the global extended header record in the archive, starting at 1.
`%%`	A `'%'` character.

Any other `'%'` characters in *string* produce undefined results.

If no −**o globexthdr.name=***string* is specified, *pax* shall use the following default value:

`$TMPDIR/GlobalHead.%n`

where $*TMPDIR* represents the value of the *TMPDIR* environment variable. If *TMPDIR* is not set, *pax* shall use **/tmp**.

invalid=*action*

(Applicable only to the −**x** *pax* format.) This keyword allows user control over the action *pax* takes upon encountering values in an extended header record that, in **read** or **copy** mode, are invalid in the destination hierarchy or, in **list** mode, cannot be written in the codeset and current locale of the implementation. The following are invalid values that shall be recognized by *pax*:

- In **read** or **copy** mode, a filename or link name that contains character encodings invalid in the destination hierarchy. (For example, the name may contain embedded NULs.)
- In **read** or **copy** mode, a filename or link name that is longer than the maximum allowed in the destination hierarchy (for either a pathname component or the entire pathname).
- In **list** mode, any character string value (filename, link name, user name, and so on) that cannot be written in the codeset and current locale of the implementation.

The following mutually-exclusive values of the *action* argument are supported:

bypass	In **read** or **copy** mode, *pax* shall bypass the file, causing no change to the destination hierarchy. In **list** mode, *pax* shall write all requested valid values for the file, but its method for writing invalid values is unspecified.
rename	In **read** or **copy** mode, *pax* shall act as if the −**i** option were in effect for each file with invalid filename or link name values, allowing the user to provide a replacement name interactively. In **list** mode, *pax* shall behave identically to the **bypass** action.
UTF-8	When used in **read**, **copy**, or **list** mode and a filename, link name, owner name, or any other field in an extended header record cannot be translated from the *pax* UTF-8 codeset format to the codeset and current locale of the implementation, *pax* shall use the actual UTF-8 encoding for the name.

write In **read** or **copy** mode, *pax* shall write the file, translating or truncating the name, regardless of whether this may overwrite an existing file with a valid name. In **list** mode, *pax* shall behave identically to the **bypass** action.

If no −**o invalid=** option is specified, *pax* shall act as if −**oinvalid=bypass** were specified. Any overwriting of existing files that may be allowed by the −**oinvalid=** actions shall be subject to permission (−**p**) and modification time (−**u**) restrictions, and shall be suppressed if the −**k** option is also specified.

linkdata
(Applicable only to the −**x** *pax* format.) In **write** mode, *pax* shall write the contents of a file to the archive even when that file is merely a hard link to a file whose contents have already been written to the archive.

listopt=*format*
This keyword specifies the output format of the table of contents produced when the −**v** option is specified in **list** mode. See **List Mode Format Specifications** (on page 2908). To avoid ambiguity, the **listopt=***format* shall be the only or final **keyword=***value* pair in a −**o** option-argument; all characters in the remainder of the option-argument shall be considered part of the format string. When multiple −**olistopt=***format* options are specified, the format strings shall be considered a single, concatenated string, evaluated in command line order.

times
(Applicable only to the −**x** *pax* format.) When used in **write** or **copy** mode, *pax* shall include **atime**, **ctime**, and **mtime** extended header records for each file. See **pax Extended Header File Times** (on page 2916).

In addition to these keywords, if the −**x** *pax* format is specified, any of the keywords and values defined in **pax Extended Header** (on page 2913), including implementation extensions, can be used in −**o** option-arguments, in either of two modes:

keyword=*value*
When used in **write** or **copy** mode, these keyword/value pairs shall be included at the beginning of the archive as **typeflag g** global extended header records. When used in **read** or **list** mode, these keyword/value pairs shall act as if they had been at the beginning of the archive as **typeflag g** global extended header records.

keyword:=*value*
When used in **write** or **copy** mode, these keyword/value pairs shall be included as records at the beginning of a **typeflag x** extended header for each file. (This shall be equivalent to the equal-sign form except that it creates no **typeflag g** global extended header records.) When used in **read** or **list** mode, these keyword/value pairs shall act as if they were included as records at the end of each extended header; thus, they shall override any global or file-specific extended header record keywords of the same names. For example, in the command:

```
pax −r −o "
gname:=mygroup,
" <archive
```

the group name will be forced to a new value for all files read from the archive.

The precedences of **–o** keywords over various fields in the archive are described in **pax Extended Header Keyword Precedence** (on page 2916).

–p *string* Specify one or more file characteristic options (privileges). The *string* option-argument shall be a string specifying file characteristics to be retained or discarded on extraction. The string shall consist of the specification characters `a`, `e`, `m`, `o`, and `p`. Other implementation-defined characters can be included. Multiple characteristics can be concatenated within the same string and multiple **–p** options can be specified. The meaning of the specification characters are as follows:

`a` Do not preserve file access times.

`e` Preserve the user ID, group ID, file mode bits (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.168, File Mode Bits), access time, modification time, and any other implementation-defined file characteristics.

`m` Do not preserve file modification times.

`o` Preserve the user ID and group ID.

`p` Preserve the file mode bits. Other implementation-defined file mode attributes may be preserved.

In the preceding list, ‘‘preserve’’ indicates that an attribute stored in the archive shall be given to the extracted file, subject to the permissions of the invoking process. The access and modification times of the file shall be preserved unless otherwise specified with the **–p** option or not stored in the archive. All attributes that are not preserved shall be determined as part of the normal file creation action (see Section 1.7.1.4 (on page 2204)).

If neither the **e** nor the **o** specification character is specified, or the user ID and group ID are not preserved for any reason, *pax* shall not set the S_ISUID and S_ISGID bits of the file mode.

If the preservation of any of these items fails for any reason, *pax* shall write a diagnostic message to standard error. Failure to preserve these items shall affect the final exit status, but shall not cause the extracted file to be deleted.

If file characteristic letters in any of the *string* option-arguments are duplicated or conflict with each other, the ones given last shall take precedence. For example, if **–p** `eme` is specified, file modification times are preserved.

–s *replstr* Modify file or archive member names named by *pattern* or *file* operands according to the substitution expression *replstr*, using the syntax of the *ed* utility. The concepts of ‘‘address’’ and ‘‘line’’ are meaningless in the context of the *pax* utility, and shall not be supplied. The format shall be:

`–s /old/new/[gp]`

where as in *ed*, *old* is a basic regular expression and *new* can contain an ampersand, `'\n'` (where *n* is a digit) backreferences, or subexpression matching. The *old* string also shall be permitted to contain <newline>s.

Any non-null character can be used as a delimiter (`'/'` shown here). Multiple **–s** expressions can be specified; the expressions shall be applied in the order specified, terminating with the first successful substitution. The optional trailing `'g'` is as defined in the *ed* utility. The optional trailing `'p'` shall cause successful substitutions to be written to standard error. File or archive member names that substitute to the empty string shall be ignored when reading and writing archives.

−t When reading files from the file system, and if the user has the permissions required by *utime*( ) to do so, set the access time of each file read to the access time that it had before being read by *pax*.

−u Ignore files that are older (having a less recent file modification time) than a pre-existing file or archive member with the same name. In **read** mode, an archive member with the same name as a file in the file system shall be extracted if the archive member is newer than the file. In **write** mode, an archive file member with the same name as a file in the file system shall be superseded if the file is newer than the archive member. If −**a** is also specified, this is accomplished by appending to the archive; otherwise, it is unspecified whether this is accomplished by actual replacement in the archive or by appending to the archive. In **copy** mode, the file in the destination hierarchy shall be replaced by the file in the source hierarchy or by a link to the file in the source hierarchy if the file in the source hierarchy is newer.

−v In **list** mode, produce a verbose table of contents (see the STDOUT section). Otherwise, write archive member pathnames to standard error (see the STDERR section).

−x *format* Specify the output archive format. The *pax* utility shall support the following formats:

cpio The *cpio* interchange format; see the EXTENDED DESCRIPTION section. The default *blocksize* for this format for character special archive files shall be 5 120. Implementations shall support all *blocksize* values less than or equal to 32 256 that are multiples of 512.

pax The *pax* interchange format; see the EXTENDED DESCRIPTION section. The default *blocksize* for this format for character special archive files shall be 5 120. Implementations shall support all *blocksize* values less than or equal to 32 256 that are multiples of 512.

ustar The *tar* interchange format; see the EXTENDED DESCRIPTION section. The default *blocksize* for this format for character special archive files shall be 10 240. Implementations shall support all *blocksize* values less than or equal to 32 256 that are multiples of 512.

Implementation-defined formats shall specify a default block size as well as any other block sizes supported for character special archive files.

Any attempt to append to an archive file in a format different from the existing archive format shall cause *pax* to exit immediately with a non-zero exit status.

In **copy** mode, if no −**x** format is specified, *pax* shall behave as if −**x***pax* were specified.

−X When traversing the file hierarchy specified by a pathname, *pax* shall not descend into directories that have a different device ID (*st_dev*; see the System Interfaces volume of IEEE Std 1003.1-200x, *stat*( )).

The options that operate on the names of files or archive members (−**c**, −**i**, −**n**, −**s**, −**u**, and −**v**) shall interact as follows. In **read** mode, the archive members shall be selected based on the user-specified *pattern* operands as modified by the −**c**, −**n**, and −**u** options. Then, any −**s** and −**i** options shall modify, in that order, the names of the selected files. The −**v** option shall write names resulting from these modifications.

In **write** mode, the files shall be selected based on the user-specified pathnames as modified by the −**n** and −**u** options. Then, any −**s** and −**i** options shall modify, in that order, the names of these selected files. The −**v** option shall write names resulting from these modifications.

If both the −**u** and −**n** options are specified, *pax* shall not consider a file selected unless it is newer than the file to which it is compared.

List Mode Format Specifications

In **list** mode with the −**o listopt=***format* option, the *format* argument shall be applied for each selected file. The *pax* utility shall append a <newline> to the **listopt** output for each selected file. The format argument shall be used as the *format* string described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation, with the exceptions 1. through 5. defined in the EXTENDED DESCRIPTION section of *printf*, plus the following exceptions:

6. The sequence (*keyword*) can occur before a format conversion specifier. The conversion argument is defined by the value of *keyword*. The implementation shall support the following keywords:

   — Any of the Field Name entries in Table 4-13 (on page 2917) and Table 4-15 (on page 2920). The implementation may support the *cpio* keywords without the leading **c_** in addition to the form required by Table 4-16 (on page 2921).

   — Any keyword defined for the extended header in **pax Extended Header** (on page 2913).

   — Any keyword provided as an implementation-defined extension within the extended header defined in **pax Extended Header** (on page 2913).

   For example, the sequence `"%(charset)s"` is the string value of the name of the character set in the extended header.

   The result of the keyword conversion argument shall be the value from the applicable header field or extended header, without any trailing NULs.

   All keyword values used as conversion arguments shall be translated from the UTF-8 encoding to the character set appropriate for the local file system, user database, and so on, as applicable.

7. An additional conversion specifier character, `T`, shall be used to specify time formats. The `T` conversion specifier character can be preceded by the sequence (*keyword*=*subformat*), where *subformat* is a date format as defined by *date* operands. The default *keyword* shall be **mtime** and the default subformat shall be:

   ```
   %b %e %H:%M %Y
   ```

8. An additional conversion specifier character, `M`, shall be used to specify the file mode string as defined in *ls* Standard Output. If (*keyword*) is omitted, the **mode** keyword shall be used. For example, `%.1M` writes the single character corresponding to the <*entry type*> field of the *ls* −**l** command.

9. An additional conversion specifier character, `D`, shall be used to specify the device for block or special files, if applicable, in an implementation-defined format. If not applicable, and (*keyword*) is specified, then this conversion shall be equivalent to `%(`*`keyword`*`)u`. If not applicable, and (*keyword*) is omitted, then this conversion shall be equivalent to <space>.

10. An additional conversion specifier character, `F`, shall be used to specify a pathname. The `F` conversion character can be preceded by a sequence of comma-separated keywords:

    ```
    (keyword[,keyword] ... )
    ```

The values for all the keywords that are non-null shall be concatenated together, each separated by a `'/'`. The default shall be (**path**) if the keyword **path** is defined; otherwise, the default shall be (**prefix**,**name**).

11. An additional conversion specifier character, `L`, shall be used to specify a symbolic line expansion. If the current file is a symbolic link, then `%L` shall expand to:

    `"%s -> %s", <value of keyword>, <contents of link>`

    Otherwise, the `%L` conversion specification shall be the equivalent of `%F`.

OPERANDS

The following operands shall be supported:

directory	The destination directory pathname for **copy** mode.
file	A pathname of a file to be copied or archived.
pattern	A pattern matching one or more pathnames of archive members. A pattern must be given in the name-generating notation of the pattern matching notation in Section 2.13 (on page 2264), including the filename expansion rules in Section 2.13.3 (on page 2265). The default, if no *pattern* is specified, is to select all members in the archive.

STDIN

In **write** mode, the standard input shall be used only if no *file* operands are specified. It shall be a text file containing a list of pathnames, one per line, without leading or trailing <blank>s.

In **list** and **read** modes, if **–f** is not specified, the standard input shall be an archive file.

Otherwise, the standard input shall not be used.

INPUT FILES

The input file named by the *archive* option-argument, or standard input when the archive is read from there, shall be a file formatted according to one of the specifications in the EXTENDED DESCRIPTION section or some other implementation-defined format.

The file **/dev/tty** shall be used to write prompts and read responses.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *pax*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements used in the pattern matching expressions for the *pattern* operand, the basic regular expression for the **–s** option, and the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the behavior of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES*

category, and pattern matching.

LC_MESSAGES
Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME Determine the format and contents of date and time strings when the −**v** option is specified.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

TMPDIR Determine the pathname that provides part of the default global extended header record file, as described for the −**o** **globexthdr=** keyword as described in the OPTIONS section.

TZ Determine the timezone used to calculate date and time strings when the −**v** option is specified. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

In **write** mode, if −**f** is not specified, the standard output shall be the archive formatted according to one of the specifications in the EXTENDED DESCRIPTION section, or some other implementation-defined format (see −**x** *format*).

In **list** mode, when the −**olistopt**=*format* has been specified, the selected archive members shall be written to standard output using the format described under **List Mode Format Specifications** (on page 2908). In **list** mode without the −**olistopt**=*format* option, the table of contents of the selected archive members shall be written to standard output using the following format: |

```
"%s\n", <path name>
```

If the −**v** option is specified in **list** mode, the table of contents of the selected archive members shall be written to standard output using the following formats.

For pathnames representing hard links to previous members of the archive:

```
"%sΔ==Δ%s\n", <ls −l listing>, <linkname>
```

For all other pathnames:

```
"%s\n", <ls −l listing>
```

where <*ls* −l *listing*> shall be the format specified by the *ls* utility with the −**l** option. When writing pathnames in this format, it is unspecified what is written for fields for which the underlying archive format does not have the correct information, although the correct number of <blank>-separated fields shall be written.

In **list** mode, standard output shall not be buffered more than a line at a time.

STDERR

If −**v** is specified in **read**, **write**, or **copy** modes, *pax* shall write the pathnames it processes to the standard error output using the following format:

```
"%s\n", <pathname>
```

These pathnames shall be written as soon as processing is begun on the file or archive member, and shall be flushed to standard error. The trailing <newline>, which shall not be buffered, is written when the file has been read or written.

If the −**s** option is specified, and the replacement string has a trailing `'p'`, substitutions shall be written to standard error in the following format:

`"%sΔ>>Δ%s\n", <original pathname>, <new pathname>`

In all operating modes of *pax*, optional messages of unspecified format concerning the input archive format and volume number, the number of files, blocks, volumes, and media parts as well as other diagnostic messages may be written to standard error.

In all formats, for both standard output and standard error, it is unspecified how non-printable characters in pathnames or link names are written.

When *pax* is in **read** mode or **list** mode, using the −**xpax** archive format, and a filename, link name, owner name, or any other field in an extended header record cannot be translated from the *pax* UTF-8 codeset format to the codeset and current locale of the implementation, *pax* shall write a diagnostic message to standard error, shall process the file as described for the −**o invalid**=option, and then shall process the next file in the archive.

OUTPUT FILES

In **read** mode, the extracted output files shall be of the archived file type. In **copy** mode, the copied output files shall be the type of the file being copied. In either mode, existing files in the destination hierarchy shall be overwritten only when all permission (−**p**), modification time (−**u**), and invalid-value (−**oinvalid**=) tests allow it.

In **write** mode, the output file named by the −**f** option-argument shall be a file formatted according to one of the specifications in the EXTENDED DESCRIPTION section, or some other implementation-defined format.

EXTENDED DESCRIPTION

pax Interchange Format

A *pax* archive tape or file produced in the −**x***pax* format shall contain a series of blocks. The physical layout of the archive shall be identical to the **ustar** format described in **ustar Interchange Format** (on page 2916). Each file archived shall be represented by the following sequence:

- An optional header block with extended header records. This header block is of the form described in **pax Header Block** (on page 2912), with a *typeflag* value of **x** or **g**. The extended header records, described in **pax Extended Header** (on page 2913), shall be included as the data for this header block.
- A header block that describes the file. Any fields in the preceding optional extended header shall override the associated fields in this header block for this file.
- Zero or more blocks that contain the contents of the file.

At the end of the archive file there shall be two 512-byte blocks filled with binary zeroes, interpreted as an end-of-archive indicator.

A schematic of an example archive with global extended header records and two actual files is shown in Figure 4-1 (on page 2912). In the example, the second file in the archive has no extended header preceding it, presumably because it has no need for extended attributes.

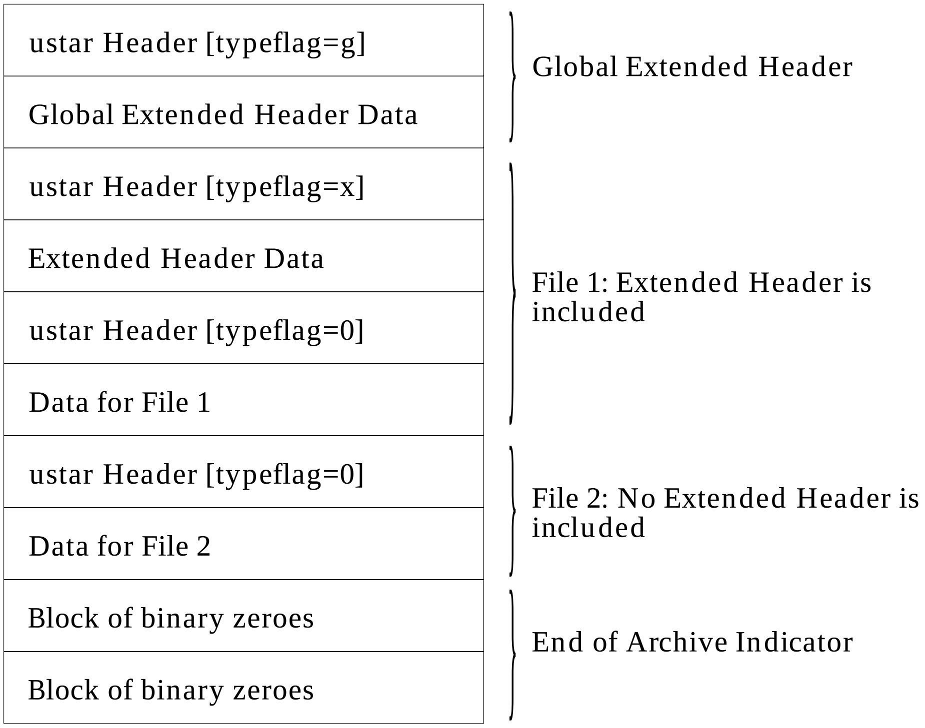


Figure 4-1 pax Format Archive Example

pax Header Block

The *pax* header block shall be identical to the **ustar** header block described in **ustar Interchange Format** (on page 2916), except that two additional *typeflag* values are defined:

x Represents extended header records for the following file in the archive (which shall have its own **ustar** header block). The format of these extended header records shall be as described in **pax Extended Header** (on page 2913).

g Represents global extended header records for the following files in the archive. The format of these extended header records shall be as described in **pax Extended Header** (on page 2913). Each value shall affect all subsequent files that do not override that value in their own extended header record and until another global extended header record is reached that provides another value for the same field. The *typeflag* **g** global headers should not be used with interchange media that could suffer partial data loss in transporting the archive.

For both of these types, the *size* field shall be the size of the extended header records in octets. The other fields in the header block are not meaningful to this version of the *pax* utility. However, if this archive is read by a *pax* utility conforming to a previous version of IEEE Std 1003.1-200x, the header block fields are used to create a regular file that contains the extended header records as data. Therefore, header block field values should be selected to provide reasonable file access to this regular file.

A further difference from the **ustar** header block is that data blocks for files of *typeflag* 1 (the digit one) (hard link) may be included, which means that the size field may be greater than zero. Archives created by **pax –o linkdata** shall include these data blocks with the hard links.

pax Extended Header

A *pax* extended header contains values that are inappropriate for the **ustar** header block because of limitations in that format: fields requiring a character encoding other than that described in the ISO/IEC 646:1991 standard, fields representing file attributes not described in the **ustar** header, and fields whose format or length do not fit the requirements of the **ustar** header. The values in an extended header add attributes to the following file (or files; see the description of the *typeflag* **g** header block) or override values in the following header block(s), as indicated in the following list of keywords.

An extended header shall consist of one or more records, each constructed as follows:

```
"%d %s=%s\n", <length>, <keyword>, <value>
```

The extended header records shall be encoded according to the ISO/IEC 10646-1:2000 standard (UTF-8). The <*length*> field, <blank>, equals sign, and <newline> shown shall be limited to the portable character set, as encoded in UTF-8. The <*keyword*> and <*value*> fields can be any UTF-8 characters. The <*length*> field shall be the decimal length of the extended header record in octets, including the trailing <newline>.

The <*keyword*> field shall be one of the entries from the following list or a keyword provided as an implementation extension. Keywords consisting entirely of lowercase letters, digits, and periods are reserved for future standardization. A keyword shall not include an equals sign. (In the following list, the notations ‘‘file(s)’’ or ‘‘block(s)’’ is used to acknowledge that a keyword affects the following single file after a *typeflag* **x** extended header, but possibly multiple files after *typeflag* **g**. Any requirements in the list for *pax* to include a record when in **write** or **copy** mode shall apply only when such a record has not already been provided through the use of the **−o** option. When used in **copy** mode, *pax* shall behave as if an archive had been created with applicable extended header records and then extracted.)

atime The file access time for the following file(s), equivalent to the value of the *st_atime* member of the **stat** structure for a file, as described by the *stat*( ) function. The access time shall be restored if the process has the appropriate privilege required to do so. The format of the <*value*> shall be as described in **pax Extended Header File Times** (on page 2916).

charset The name of the character set used to encode the data in the following file(s). The entries in the following table are defined to refer to known standards; additional names may be agreed on between the originator and recipient.

<value>	Formal Standard
`ISO-IRΔ646Δ1990`	ISO / IEC 646: 1990
`ISO-IRΔ8859Δ1Δ1998`	ISO / IEC 8859-1: 1998
`ISO-IRΔ8859Δ2Δ1999`	ISO / IEC 8859-2: 1999
`ISO-IRΔ8859Δ3Δ1999`	ISO / IEC 8859-3: 1999
`ISO-IRΔ8859Δ4Δ1998`	ISO / IEC 8859-4: 1998
`ISO-IRΔ8859Δ5Δ1999`	ISO / IEC 8859-5: 1999
`ISO-IRΔ8859Δ6Δ1999`	ISO / IEC 8859-6: 1999
`ISO-IRΔ8859Δ7Δ1987`	ISO / IEC 8859-7: 1987
`ISO-IRΔ8859Δ8Δ1999`	ISO / IEC 8859-8: 1999
`ISO-IRΔ8859Δ9Δ1999`	ISO / IEC 8859-9: 1999
`ISO-IRΔ8859Δ10Δ1998`	ISO / IEC 8859-10: 1998
`ISO-IRΔ8859Δ13Δ1998`	ISO / IEC 8859-13: 1998
`ISO-IRΔ8859Δ14Δ1998`	ISO / IEC 8859-14: 1998
`ISO-IRΔ8859Δ15Δ1999`	ISO / IEC 8859-15: 1999
`ISO-IRΔ10646Δ2000`	ISO / IEC 10646: 2000
`ISO-IRΔ10646Δ2000ΔUTF-8`	ISO / IEC 10646, UTF-8 encoding
`BINARY`	None.

The encoding is included in an extended header for information only; when *pax* is used as described in IEEE Std 1003.1-200x, it shall not translate the file data into any other encoding. The **BINARY** entry indicates unencoded binary data.

When used in **write** or **copy** mode, it is implementation-defined whether *pax* includes a **charset** extended header record for a file.

comment A series of characters used as a comment. All characters in the <*value*> field shall be ignored by *pax*.

ctime The file creation time for the following file(s), equivalent to the value of the *st_ctime* member of the **stat** structure for a file, as described by the *stat*( ) function. The creation time shall be restored if the process has the appropriate privilege required to do so. The format of the <*value*> shall be as described in **pax Extended Header File Times** (on page 2916).

gid The group ID of the group that owns the file, expressed as a decimal number using digits from the ISO / IEC 646: 1991 standard. This record shall override the *gid* field in the following header block(s). When used in **write** or **copy** mode, *pax* shall include a *gid* extended header record for each file whose group ID is greater than 2 097 151 (octal 7 777 777).

gname The group of the file(s), formatted as a group name in the group database. This record shall override the *gid* and *gname* fields in the following header block(s), and any *gid* extended header record. When used in **read**, **copy**, or **list** mode, *pax* shall translate the name from the UTF-8 encoding in the header record to the character set appropriate for the group database on the receiving system. If any of the UTF-8 characters cannot be translated, and if the −**oinvalid=**UTF-8 option is not specified, the results are implementation-defined. When used in **write** or **copy** mode, *pax* shall include a **gname** extended header record for each file whose group name cannot be represented entirely with the letters and digits of the portable character set.

linkpath The pathname of a link being created to another file, of any type, previously archived. This record shall override the *linkname* field in the following **ustar** header block(s). The following **ustar** header block shall determine the type of link created.

	If *typeflag* of the following header block is 1, it shall be a hard link. If *typeflag* is 2, it shall be a symbolic link and the **linkpath** value shall be the contents of the symbolic link. The *pax* utility shall translate the name of the link (contents of the symbolic link) from the UTF-8 encoding to the character set appropriate for the local file system. When used in **write** or **copy** mode, *pax* shall include a **linkpath** extended header record for each link whose pathname cannot be represented entirely with the members of the portable character set other than NUL.
mtime	The file modification time of the following file(s), equivalent to the value of the *st_mtime* member of the **stat** structure for a file, as described in the *stat*( ) function. This record shall override the *mtime* field in the following header block(s). The modification time shall be restored if the process has the appropriate privilege required to do so. The format of the <*value*> shall be as described in **pax Extended Header File Times** (on page 2916).
path	The pathname of the following file(s). This record shall override the *name* and *prefix* fields in the following header block(s). The *pax* utility shall translate the pathname of the file from the UTF-8 encoding to the character set appropriate for the local file system. When used in **write** or **copy** mode, *pax* shall include a *path* extended header record for each file whose pathname cannot be represented entirely with the members of the portable character set other than NUL.
realtime.*any*	The keywords prefixed by ‘‘realtime.’’ are reserved for future standardization.
security.*any*	The keywords prefixed by ‘‘security.’’ are reserved for future standardization.
size	The size of the file in octets, expressed as a decimal number using digits from the ISO/IEC 646:1991 standard. This record shall override the *size* field in the following header block(s). When used in **write** or **copy** mode, *pax* shall include a *size* extended header record for each file with a size value greater than 8 589 934 591 (octal 7 777 777 777).
uid	The user ID of the file owner, expressed as a decimal number using digits from the ISO/IEC 646:1991 standard. This record shall override the *uid* field in the following header block(s). When used in **write** or **copy** mode, *pax* shall include a *uid* extended header record for each file whose owner ID is greater than 2 097 151 (octal 7 777 777).
uname	The owner of the following file(s), formatted as a user name in the user database. This record shall override the *uid* and *uname* fields in the following header block(s), and any *uid* extended header record. When used in **read**, **copy**, or **list** mode, *pax* shall translate the name from the UTF-8 encoding in the header record to the character set appropriate for the user database on the receiving system. If any of the UTF-8 characters cannot be translated, and if the –**oinvalid=** UTF-8 option is not specified, the results are implementation-defined. When used in **write** or **copy** mode, *pax* shall include a **uname** extended header record for each file whose user name cannot be represented entirely with the letters and digits of the portable character set.

If the <*value*> field is zero length, it shall delete any header block field, previously entered extended header value, or global extended header value of the same name.

If a keyword in an extended header record (or in a –**o** option-argument) overrides or deletes a corresponding field in the **ustar** header block, *pax* shall ignore the contents of that header block field.

Unlike the **ustar** header block fields, NULs shall not delimit <*value*>s; all characters within the <*value*> field shall be considered data for the field. None of the length limitations of the **ustar** header block fields in Table 4-13 (on page 2917) shall apply to the extended header records.

pax Extended Header Keyword Precedence

This section describes the precedence in which the various header records and fields and command line options are selected to apply to a file in the archive. When *pax* is used in **read** or **list** modes, it shall determine a file attribute in the following sequence:

1. If −**odelete=***keyword-prefix* is used, the affected attributes shall be determined from step 7., if applicable, or ignored otherwise.
2. If −**o***keyword*:= is used, the affected attributes shall be ignored.
3. If −**o***keyword***:=***value* is used, the affected attribute shall be assigned the value.
4. If there is a *typeflag* **x** extended header record, the affected attribute shall be assigned the <*value*>. When extended header records conflict, the last one given in the header shall take precedence.
5. If −**o***keyword***=***value* is used, the affected attribute shall be assigned the value.
6. If there is a *typeflag* **g** global extended header record, the affected attribute shall be assigned the <*value*>. When global extended header records conflict, the last one given in the global header shall take precedence.
7. Otherwise, the attribute shall be determined from the **ustar** header block.

pax Extended Header File Times

The *pax* utility shall write an **mtime** record for each file in **write** or **copy** modes if the file's modification time cannot be represented exactly in the **ustar** header logical record described in **ustar Interchange Format**. This can occur if the time is out of **ustar** range, or if the file system of the underlying implementation supports non-integer time granularities and the time is not an integer. All of these time records shall be formatted as a decimal representation of the time in seconds since the Epoch. If a period (`'.'`) decimal point character is present, the digits to the right of the point shall represent the units of a subsecond timing granularity, where the first digit is tenths of a second and each subsequent digit is a tenth of the previous digit. In **read** or **copy** mode, the *pax* utility shall truncate the time of a file to the greatest value that is not greater than the input header file time. In **write** or **copy** mode, the *pax* utility shall output a time exactly if it can be represented exactly as a decimal number, and otherwise shall generate only enough digits so that the same time shall be recovered if the file is extracted on a system whose underlying implementation supports the same time granularity.

ustar Interchange Format

A **ustar** archive tape or file shall contain a series of logical records. Each logical record shall be a fixed-size logical record of 512 octets (see below). Although this format may be thought of as being stored on 9-track industry-standard 12.7mm (0.5in) magnetic tape, other types of transportable media are not excluded. Each file archived shall be represented by a header logical record that describes the file, followed by zero or more logical records that give the contents of the file. At the end of the archive file there shall be two 512-octet logical records filled with binary zeros, interpreted as an end-of-archive indicator.

The logical records may be grouped for physical I/O operations, as described under the −**b***blocksize* and −**x** **ustar** options. Each group of logical records may be written with a single operation equivalent to the *write*( ) function. On magnetic tape, the result of this write shall be a single tape physical block. The last physical block shall always be the full size, so logical records after the two zero logical records may contain undefined data.

The header logical record shall be structured as shown in the following table. All lengths and offsets are in decimal.

Table 4-13 ustar Header Block

Field Name	**Octet Offset**	**Length (in Octets)**
name	0	100
mode	100	8
uid	108	8
gid	116	8
size	124	12
mtime	136	12
chksum	148	8
typeflag	156	1
linkname	157	100
magic	257	6
version	263	2
uname	265	32
gname	297	32
devmajor	329	8
devminor	337	8
prefix	345	155

All characters in the header logical record shall be represented in the coded character set of the ISO/IEC 646:1991 standard. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as octets with the most significant bit zero. If an implementation supports the use of characters outside of slash and the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes.

However, the *pax* utility shall never create filenames on the local system that cannot be accessed via the procedures described in IEEE Std 1003.1-200x. If a filename is found on the medium that would create an invalid filename, it is implementation-defined whether the data from the file is stored on the file hierarchy and under what name it is stored. The *pax* utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

Each field within the header logical record is contiguous; that is, there is no padding used. Each character on the archive medium shall be stored contiguously.

The fields *magic*, *uname*, and *gname* are character strings each terminated by a NUL character. The fields *name*, *linkname*, and *prefix* are NUL-terminated character strings except when all characters in the array contain non-NUL characters including the last character. The *version* field is two octets containing the characters `"00"` (zero-zero). The *typeflag* contains a single character. All other fields are leading zero-filled octal numbers using digits from the ISO/IEC 646:1991 standard IRV. Each numeric field is terminated by one or more <space> or NUL characters.

The *name* and the *prefix* fields shall produce the pathname of the file. A new pathname shall be formed, if *prefix* is not an empty string (its first character is not NUL), by concatenating *prefix* (up to the first NUL character), a slash character, and *name*; otherwise, *name* is used alone. In either case, *name* is terminated at the first NUL character. If *prefix* begins with a NUL character, it shall be ignored. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, *pax* shall notify the user of the error, and shall not store any part of the file—header or data—on the medium.

The *linkname* field, described below, shall not use the *prefix* to produce a pathname. As such, a *linkname* is limited to 100 characters. If the name does not fit in the space provided, *pax* shall notify the user of the error, and shall not attempt to store the link on the medium.

The *mode* field provides 12 bits encoded in the ISO/IEC 646: 1991 standard octal digit representation. The encoded bits shall represent the following values:

Table 4-14 ustar *mode* Field

Bit Value	IEEE Std 1003.1-200x Bit	Description
04 000	S_ISUID	Set UID on execution.
02 000	S_ISGID	Set GID on execution.
01 000	<reserved>	Reserved for future standardization.
00 400	S_IRUSR	Read permission for file owner class.
00 200	S_IWUSR	Write permission for file owner class.
00 100	S_IXUSR	Execute/search permission for file owner class.
00 040	S_IRGRP	Read permission for file group class.
00 020	S_IWGRP	Write permission for file group class.
00 010	S_IXGRP	Execute/search permission for file group class.
00 004	S_IROTH	Read permission for file other class.
00 002	S_IWOTH	Write permission for file other class.
00 001	S_IXOTH	Execute/search permission for file other class.

When appropriate privilege is required to set one of these mode bits, and the user restoring the files from the archive does not have the appropriate privilege, the mode bits for which the user does not have appropriate privilege shall be ignored. Some of the mode bits in the archive format are not mentioned elsewhere in this volume of IEEE Std 1003.1-200x. If the implementation does not support those bits, they may be ignored.

The *uid* and *gid* fields are the user and group ID of the owner and group of the file, respectively.

The *size* field is the size of the file in octets. If the *typeflag* field is set to specify a file to be of type 1 (a link) or 2 (a symbolic link), the *size* field shall be specified as zero. If the *typeflag* field is set to specify a file of type 5 (directory), the *size* field shall be interpreted as described under the definition of that record type. No data logical records are stored for types 1, 2, or 5. If the *typeflag* field is set to 3 (character special file), 4 (block special file), or 6 (FIFO), the meaning of the *size* field is unspecified by this volume of IEEE Std 1003.1-200x, and no data logical records shall be stored on the medium. Additionally, for type 6, the *size* field shall be ignored when reading. If the *typeflag* field is set to any other value, the number of logical records written following the header shall be (*size*+511)/512, ignoring any fraction in the result of the division. |

The *mtime* field shall be the modification time of the file at the time it was archived. It is the ISO/IEC 646: 1991 standard representation of the octal value of the modification time obtained from the *stat*( ) function.

The *chksum* field shall be the ISO/IEC 646: 1991 standard IRV representation of the octal value of the simple sum of all octets in the header logical record. Each octet in the header shall be treated as an unsigned value. These values shall be added to an unsigned integer, initialized to zero, the precision of which is not less than 17 bits. When calculating the checksum, the *chksum* field is treated as if it were all spaces.

The *typeflag* field specifies the type of file archived. If a particular implementation does not recognize the type, or the user does not have appropriate privilege to create that type, the file shall be extracted as if it were a regular file if the file type is defined to have a meaning for the *size* field that could cause data logical records to be written on the medium (see the previous description for *size*). If conversion to a regular file occurs, the *pax* utility shall produce an error indicating that the conversion took place. All of the *typeflag* fields shall be coded in the ISO/IEC 646:1991 standard IRV:

`0` Represents a regular file. For backward compatibility, a *typeflag* value of binary zero (`'\0'`) should be recognized as meaning a regular file when extracting files from the archive. Archives written with this version of the archive file format create regular files with a *typeflag* value of the ISO/IEC 646:1991 standard IRV `'0'`.

`1` Represents a file linked to another file, of any type, previously archived. Such files are identified by each file having the same device and file serial number. The linked-to name is specified in the *linkname* field with a NUL-character terminator if it is less than 100 octets in length.

`2` Represents a symbolic link. The contents of the symbolic link shall be stored in the *linkname* field.

`3,4` Represent character special files and block special files respectively. In this case the *devmajor* and *devminor* fields shall contain information defining the device, the format of which is unspecified by this volume of IEEE Std 1003.1-200x. Implementations may map the device specifications to their own local specification or may ignore the entry.

`5` Specifies a directory or subdirectory. On systems where disk allocation is performed on a directory basis, the *size* field shall contain the maximum number of octets (which may be rounded to the nearest disk block allocation unit) that the directory may hold. A *size* field of zero indicates no such limiting. Systems that do not support limiting in this manner should ignore the *size* field.

`6` Specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.

`7` Reserved to represent a file to which an implementation has associated some high-performance attribute. Implementations without such extensions should treat this file as a regular file (type 0).

`A-Z` The letters `'A'` to `'Z'`, inclusive, are reserved for custom implementations. All other values are reserved for future versions of IEEE Std 1003.1-200x.

Attempts to archive a socket using **ustar** interchange format shall produce a diagnostic message. Handling of other file types is implementation-defined.

The *magic* field is the specification that this archive was output in this archive format. If this field contains **ustar** (the five characters from the ISO/IEC 646:1991 standard IRV shown followed by NUL), the *uname* and *gname* fields shall contain the ISO/IEC 646:1991 standard IRV representation of the owner and group of the file, respectively (truncated to fit, if necessary). When the file is restored by a privileged, protection-preserving version of the utility, the user and group databases shall be scanned for these names. If found, the user and group IDs contained within these files shall be used rather than the values contained within the *uid* and *gid* fields.

cpio Interchange Format

The octet-oriented *cpio* archive format shall be a series of entries, each comprising a header that describes the file, the name of the file, and then the contents of the file.

An archive may be recorded as a series of fixed-size blocks of octets. This blocking shall be used only to make physical I/O more efficient. The last group of blocks shall be always at the full size.

For the octet-oriented *cpio* archive format, the individual entry information shall be in the order indicated and described by the following table; see also the **<cpio.h>** header.

Table 4-15 Octet-Oriented *cpio* Archive Entry

Header Field Name	**Length (in Octets)**	**Interpreted as**
c_magic	6	Octal number
c_dev	6	Octal number
c_ino	6	Octal number
c_mode	6	Octal number
c_uid	6	Octal number
c_gid	6	Octal number
c_nlink	6	Octal number
c_rdev	6	Octal number
c_mtime	11	Octal number
c_namesize	6	Octal number
c_filesize	11	Octal number
Filename Field Name	**Length**	**Interpreted as**
c_name	*c_namesize*	Pathname string
File Data Field Name	**Length**	**Interpreted as**
c_filedata	*c_filesize*	Data

cpio Header

For each file in the archive, a header as defined previously shall be written. The information in the header fields is written as streams of the ISO/IEC 646:1991 standard characters interpreted as octal numbers. The octal numbers shall be extended to the necessary length by appending the ISO/IEC 646:1991 standard IRV zeros at the most-significant-digit end of the number; the result is written to the most-significant digit of the stream of octets first. The fields shall be interpreted as follows:

c_magic Identify the archive as being a transportable archive by containing the identifying value `"070707"`.

c_dev, *c_ino* Contains values that uniquely identify the file within the archive (that is, no files contain the same pair of *c_dev* and *c_ino* values unless they are links to the same file). The values shall be determined in an unspecified manner.

c_mode Contains the file type and access permissions as defined in the following table.

Table 4-16 Values for *cpio c_mode* Field

File Permissions Name	Value	Indicates
C_IRUSR	000 400	Read by owner
C_IWUSR	000 200	Write by owner
C_IXUSR	000 100	Execute by owner
C_IRGRP	000 040	Read by group
C_IWGRP	000 020	Write by group
C_IXGRP	000 010	Execute by group
C_IROTH	000 004	Read by others
C_IWOTH	000 002	Write by others
C_IXOTH	000 001	Execute by others
C_ISUID	004 000	Set *uid*
C_ISGID	002 000	Set *gid*
C_ISVTX	001 000	Reserved
File Type Name	**Value**	**Indicates**
C_ISDIR	040 000	Directory
C_ISFIFO	010 000	FIFO
C_ISREG	0100 000	Regular file
C_ISLNK	0120 000	Symbolic link
C_ISBLK	060 000	Block special file
C_ISCHR	020 000	Character special file
C_ISSOCK	0140 000	Socket
C_ISCTG	0110 000	Reserved

Directories, FIFOs, symbolic links, and regular files shall be supported on a system conforming to this volume of IEEE Std 1003.1-200x; additional values defined previously are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written to archives intended to be transported to other systems.

c_uid	Contains the user ID of the owner.
c_gid	Contains the group ID of the group.
c_nlink	Contains the number of links referencing the file at the time the archive was created.
c_rdev	Contains implementation-defined information for character or block special files.
c_mtime	Contains the latest time of modification of the file at the time the archive was created.
c_namesize	Contains the length of the pathname, including the terminating NUL character.
c_filesize	Contains the length of the file in octets. This shall be the length of the data section following the header structure.

cpio Filename

The *c_name* field shall contain the pathname of the file. The length of this field in octets is the value of *c_namesize*.

If a filename is found on the medium that would create an invalid pathname, it is implementation-defined whether the data from the file is stored on the file hierarchy and under what name it is stored.

All characters shall be represented in the ISO/IEC 646:1991 standard IRV. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as octets with the most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes. However, the *pax* utility shall never create filenames on the local system that cannot be accessed via the procedures described previously in this volume of IEEE Std 1003.1-200x. If a filename is found on the medium that would create an invalid filename, it is implementation-defined whether the data from the file is stored on the local file system and under what name it is stored. The *pax* utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

cpio File Data

Following *c_name*, there shall be *c_filesize* octets of data. Interpretation of such data occurs in a manner dependent on the file. If *c_filesize* is zero, no data shall be contained in *c_filedata*.

When restoring from an archive:

- If the user does not have the appropriate privilege to create a file of the specified type, *pax* shall ignore the entry and write an error message to standard error.
- Only regular files have data to be restored. Presuming a regular file meets any selection criteria that might be imposed on the format-reading utility by the user, such data shall be restored.
- If a user does not have appropriate privilege to set a particular mode flag, the flag shall be ignored. Some of the mode flags in the archive format are not mentioned elsewhere in this volume of IEEE Std 1003.1-200x. If the implementation does not support those flags, they may be ignored.

cpio Special Entries

FIFO special files, directories, and the trailer shall be recorded with *c_filesize* equal to zero. For other special files, *c_filesize* is unspecified by this volume of IEEE Std 1003.1-200x. The header for the next file entry in the archive shall be written directly after the last octet of the file entry preceding it. A header denoting the filename **TRAILER!!!** shall indicate the end of the archive; the contents of octets in the last block of the archive following such a header are undefined. |

EXIT STATUS

The following exit values shall be returned:

0	All files were processed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If *pax* cannot create a file or a link when reading an archive or cannot find a file when writing an archive, or cannot preserve the user ID, group ID, or file mode when the −**p** option is specified, a diagnostic message shall be written to standard error and a non-zero exit status shall be returned, but processing shall continue. In the case where *pax* cannot create a link to a file, *pax* shall not, by default, create a second copy of the file.

If the extraction of a file from an archive is prematurely terminated by a signal or error, *pax* may have only partially extracted the file or (if the −**n** option was not specified) may have extracted a file of the same name as that specified by the user, but which is not the file the user wanted. Additionally, the file modes of extracted directories may have additional bits from the S_IRWXU mask set as well as incorrect modification and access times.

APPLICATION USAGE

The −**p** (privileges) option was invented to reconcile differences between historical *tar* and *cpio* implementations. In particular, the two utilities use −**m** in diametrically opposed ways. The −**p** option also provides a consistent means of extending the ways in which future file attributes can be addressed, such as for enhanced security systems or high-performance files. Although it may seem complex, there are really two modes that are most commonly used:

−**p e**	''Preserve everything''. This would be used by the historical superuser, someone with all the appropriate privileges, to preserve all aspects of the files as they are recorded in the archive. The **e** flag is the sum of **o** and **p**, and other implementation-defined attributes.
−**p p**	''Preserve'' the file mode bits. This would be used by the user with regular privileges who wished to preserve aspects of the file other than the ownership. The file times are preserved by default, but two other flags are offered to disable these and use the time of extraction.

The one pathname per line format of standard input precludes pathnames containing <newline>s. Although such pathnames violate the portable filename guidelines, they may exist and their presence may inhibit usage of *pax* within shell scripts. This problem is inherited from historical archive programs. The problem can be avoided by listing filename arguments on the command line instead of on standard input.

It is almost certain that appropriate privileges are required for *pax* to accomplish parts of this volume of IEEE Std 1003.1-200x. Specifically, creating files of type block special or character special, restoring file access times unless the files are owned by the user (the −**t** option), or preserving file owner, group, and mode (the −**p** option) all probably require appropriate privileges.

In **read** mode, implementations are permitted to overwrite files when the archive has multiple members with the same name. This may fail if permissions on the first version of the file do not permit it to be overwritten.

The **cpio** and **ustar** formats can only support files up to 8589934592 bytes (8 ∗ 2ˆ30) in size.

EXAMPLES

The following command:

```
pax −w −f /dev/rmt/1m .
```

copies the contents of the current directory to tape drive 1, medium density (assuming historical System V device naming procedures. The historical BSD device name would be **/dev/rmt9**).

The following commands:

```
mkdir newdir
pax –rw olddir newdir
```

copy the *olddir* directory hierarchy to *newdir*.

```
pax –r –s ',ˆ//*usr//*,,' –f a.pax
```

reads the archive **a.pax**, with all files rooted in **/usr** in the archive extracted relative to the current directory.

Using the option:

```
–o listopt="%M %(atime)T %(size)D %(name)s"
```

overrides the default output description in Standard Output and instead writes:

```
–rw–rw––– Jan 12 15:53 1492 /usr/foo/bar
```

Using the options:

```
–o listopt='%L\t%(size)D\n%.7' \
–o listopt='(name)s\n%(ctime)T\n%T'
```

overrides the default output description in Standard Output and instead writes:

```
/usr/foo/bar –> /tmp   1492
/usr/fo
Jan 12 1991
Jan 31 15:53
```

RATIONALE

The *pax* utility was new, commissioned for the ISO POSIX-2: 1993 standard. It represents a peaceful compromise between advocates of the historical *tar* and *cpio* utilities.

A fundamental difference between *cpio* and *tar* was in the way directories were treated. The *cpio* utility did not treat directories differently from other files, and to select a directory and its contents required that each file in the hierarchy be explicitly specified. For *tar*, a directory matched every file in the file hierarchy it rooted.

The *pax* utility offers both interfaces; by default, directories map into the file hierarchy they root. The **–d** option causes *pax* to skip any file not explicitly referenced, as *cpio* historically did. The *tar –style* behavior was chosen as the default because it was believed that this was the more common usage and because *tar* is the more commonly available interface, as it was historically provided on both System V and BSD implementations.

The data interchange format specification in this volume of IEEE Std 1003.1-200x requires that processes with ''appropriate privileges'' shall always restore the ownership and permissions of extracted files exactly as archived. If viewed from the historic equivalence between superuser and ''appropriate privileges'', there are two problems with this requirement. First, users running as superusers may unknowingly set dangerous permissions on extracted files. Second, it is needlessly limiting, in that superusers cannot extract files and own them as superuser unless the archive was created by the superuser. (It should be noted that restoration of ownerships and permissions for the superuser, by default, is historical practice in *cpio*, but not in *tar*.) In order to avoid these two problems, the *pax* specification has an additional ''privilege'' mechanism, the **–p** option. Only a *pax* invocation with the privileges needed, and which has the **–p** option set using the **e** specification character, has the ''appropriate privilege'' to restore full ownership and permission information.

Note also that this volume of IEEE Std 1003.1-200x requires that the file ownership and access permissions shall be set, on extraction, in the same fashion as the *creat*( ) function when provided the mode stored in the archive. This means that the file creation mask of the user is applied to the file permissions.

Users should note that directories may be created by *pax* while extracting files with permissions that are different from those that existed at the time the archive was created. When extracting sensitive information into a directory hierarchy that no longer exists, users are encouraged to set their file creation mask appropriately to protect these files during extraction.

The table of contents output is written to standard output to facilitate pipeline processing.

An early proposal had hard links displaying for all pathnames. This was removed because it complicates the output of the case where −**v** is not specified and does not match historical *cpio* usage. The hard-link information is available in the −**v** display.

The description of the −**l** option allows implementations to make hard links to symbolic links. IEEE Std 1003.1-200x does not specify any way to create a hard link to a symbolic link, but many implementations provide this capability as an extension. If there are hard links to symbolic links when an archive is created, the implementation is required to archive the hard link in the archive (unless −**H** or −**L** is specified). When in **read** mode and in **copy** mode, implementations supporting hard links to symbolic links should use them when appropriate.

The archive formats inherited from the POSIX.1-1990 standard have certain restrictions that have been brought along from historical usage. For example, there are restrictions on the length of pathnames stored in the archive. When *pax* is used in **copy**(−**rw**) mode (copying directory hierarchies), the ability to use extensions from the −**x***pax* format overcomes these restrictions.

The default *blocksize* value of 5 120 bytes for *cpio* was selected because it is one of the standard block-size values for *cpio*, set when the −**B** option is specified. (The other default block-size value for *cpio* is 512 bytes, and this was considered to be too small.) The default block value of 10 240 bytes for *tar* was selected because that is the standard block-size value for BSD *tar*. The maximum block size of 32 256 bytes (2^{15}−512 bytes) is the largest multiple of 512 bytes that fits into a signed 16-bit tape controller transfer register. There are known limitations in some historical systems that would prevent larger blocks from being accepted. Historical values were chosen to improve compatibility with historical scripts using *dd* or similar utilities to manipulate archives. Also, default block sizes for any file type other than character special file has been deleted from this volume of IEEE Std 1003.1-200x as unimportant and not likely to affect the structure of the resulting archive.

Implementations are permitted to modify the block-size value based on the archive format or the device to which the archive is being written. This is to provide implementations with the opportunity to take advantage of special types of devices, and it should not be used without a great deal of consideration as it almost certainly decreases archive portability.

The intended use of the −**n** option was to permit extraction of one or more files from the archive without processing the entire archive. This was viewed by the standard developers as offering significant performance advantages over historical implementations. The −**n** option in early proposals had three effects; the first was to cause special characters in patterns to not be treated specially. The second was to cause only the first file that matched a pattern to be extracted. The third was to cause *pax* to write a diagnostic message to standard error when no file was found matching a specified pattern. Only the second behavior is retained by this volume of IEEE Std 1003.1-200x, for many reasons. First, it is in general not acceptable for a single option to have multiple effects. Second, the ability to make pattern matching characters act as normal characters is useful for parts of *pax* other than file extraction. Third, a finer degree of control over the special characters is useful because users may wish to normalize only a single special character in a single filename. Fourth, given a more general escape mechanism, the previous behavior of the −**n** option can be easily obtained using the −**s** option or a *sed* script. Finally, writing a diagnostic message when a pattern specified by the user is unmatched by any file is useful behavior in all cases.

In this version, the −**n** was removed from the **copy** mode synopsis of *pax*; it is inapplicable because there are no pattern operands specified in this mode.

There is another method than *pax* for copying subtrees in IEEE Std 1003.1-200x described as part of the *cp* utility. Both methods are historical practice: *cp* provides a simpler, more intuitive interface, while *pax* offers a finer granularity of control. Each provides additional functionality to the other; in particular, *pax* maintains the hard-link structure of the hierarchy while *cp* does not. It is the intention of the standard developers that the results be similar (using appropriate option combinations in both utilities). The results are not required to be identical; there seemed insufficient gain to applications to balance the difficulty of implementations having to guarantee that the results would be exactly identical.

A single archive may span more than one file. It is suggested that implementations provide informative messages to the user on standard error whenever the archive file is changed.

The −**d** option (do not create intermediate directories not listed in the archive) found in early proposals was originally provided as a complement to the historic −**d** option of *cpio*. It has been deleted.

The −**s** option in early proposals specified a subset of the substitution command from the *ed* utility. As there was no reason for only a subset to be supported, the −**s** option is now compatible with the current *ed* specification. Since the delimiter can be any non-null character, the following usage with single spaces is valid:

```
pax −s " foo bar " ...
```

The −**t** description is worded so as to note that this may cause the access time update caused by some other activity (which occurs while the file is being read) to be overwritten.

The default behavior of *pax* with regard to file modification times is the same as historical implementations of *tar*. It is not the historical behavior of *cpio*.

Because the −**i** option uses **/dev/tty**, utilities without a controlling terminal are not able to use this option.

The −**y** option, found in early proposals, has been deleted because a line containing a single period for the −**i** option has equivalent functionality. The special lines for the −**i** option (a single period and the empty line) are historical practice in *cpio*.

In early drafts, an −**e***charmap* option was included to increase portability of files between systems using different coded character sets. This option was omitted because it was apparent that consensus could not be formed for it. In this version, the use of UTF-8 should be an adequate substitute.

The −**k** option was added to address international concerns about the dangers involved in the character set transformations of −**e** (if the target character set were different from the source, the filenames might be transformed into names matching existing files) and also was made more general to protect files transferred between file systems with different {NAME_MAX} values (truncating a filename on a smaller system might also inadvertently overwrite existing files). As stated, it prevents any overwriting, even if the target file is older than the source. This version adds more granularity of options to solve this problem by introducing the −**oinvalid=** option—specifically the UTF-8 action. (Note that an existing file that is named with a UTF-8 encoding is still subject to overwriting in this case. The −**k** option closes that loophole.)

Some of the file characteristics referenced in this volume of IEEE Std 1003.1-200x might not be supported by some archive formats. For example, neither the *tar* nor *cpio* formats contain the file access time. For this reason, the **e** specification character has been provided, intended to cause all file characteristics specified in the archive to be retained.

It is required that extracted directories, by default, have their access and modification times and permissions set to the values specified in the archive. This has obvious problems in that the directories are almost certainly modified after being extracted and that directory permissions may not permit file creation. One possible solution is to create directories with the mode specified in the archive, as modified by the *umask* of the user, with sufficient permissions to allow file creation. After all files have been extracted, *pax* would then reset the access and modification times and permissions as necessary.

The list-mode formatting description borrows heavily from the one defined by the *printf* utility. However, since there is no separate operand list to get conversion arguments, the format was extended to allow specifying the name of the conversion argument as part of the conversion specification.

The `T` conversion specifier allows time fields to be displayed in any of the date formats. Unlike the *ls* utility, *pax* does not adjust the format when the date is less than six months in the past. This makes parsing the output more predictable.

The `D` conversion specifier handles the ability to display the major/minor or file size, as with *ls*, by using `%−8(`*`size`*`)D`.

The `L` conversion specifier handles the *ls* display for symbolic links.

Conversion specifiers were added to generate existing known types used for *ls*.

pax Interchange Format

The new POSIX data interchange format was developed primarily to satisfy international concerns that the **ustar** and *cpio* formats did not provide for file, user, and group names encoded in characters outside a subset of the ISO/IEC 646:1991 standard. The standard developers realized that this new POSIX data interchange format should be very extensible because there were other requirements they foresaw in the near future:

- Support international character encodings and locale information
- Support security information (ACLs, and so on)
- Support future file types, such as realtime or contiguous files
- Include data areas for implementation use
- Support systems with words larger than 32 bits and timers with subsecond granularity

The following were not goals for this format because these are better handled by separate utilities or are inappropriate for a portable format:

- Encryption
- Compression
- Data translation between locales and codesets
- *inode* storage

The format chosen to support the goals is an extension of the **ustar** format. Of the two formats previously available, only the **ustar** format was selected for extensions because:

- It was easier to extend in an upward-compatible way. It offered version flags and header block type fields with room for future standardization. The *cpio* format, while possessing a more flexible file naming methodology, could not be extended without breaking some theoretical implementation or using a dummy filename that could be a legitimate filename.

- Industry experience since the original ''tar wars'' fought in developing the ISO POSIX-1 standard has clearly been in favor of the **ustar** format, which is generally the default output format selected for *pax* implementations on new systems.

The new format was designed with one additional goal in mind: reasonable behavior when an older *tar* or *pax* utility happened to read an archive. Since the POSIX.1-1990 standard mandated that a ''format-reading utility'' had to treat unrecognized *typeflag* values as regular files, this allowed the format to include all the extended information in a pseudo-regular file that preceded each real file. An option is given that allows the archive creator to set up reasonable names for these files on the older systems. Also, the normative text suggests that reasonable file access values be used for this **ustar** header block. Making these header files inaccessible for convenient reading and deleting would not be reasonable. File permissions of 600 or 700 are suggested.

The **ustar** *typeflag* field was used to accommodate the additional functionality of the new format rather than magic or version because the POSIX.1-1990 standard (and, by reference, the previous version of *pax*), mandated the behavior of the format-reading utility when it encountered an unknown *typeflag*, but was silent about the other two fields.

Early proposals of the first revision to IEEE Std 1003.1-200x contained a proposed archive format that was based on compatibility with the standard for tape files (ISO 1001, similar to the format used historically on many mainframes and minicomputers). This format was overly complex and required considerable overhead in volume and header records. Furthermore, the standard developers felt that it would not be acceptable to the community of POSIX developers, so it was later changed to be a format more closely related to historical practice on POSIX systems.

The prefix and name split of pathnames in **ustar** was replaced by the single path extended header record for simplicity.

The concept of a global extended header (*typeflag* **g**) was controversial. If this were applied to an archive being recorded on magnetic tape, a few unreadable blocks at the beginning of the tape could be a serious problem; a utility attempting to extract as many files as possible from a damaged archive could lose a large percentage of file header information in this case. However, if the archive were on a reliable medium, such as a CD-ROM, the global extended header offers considerable potential size reductions by eliminating redundant information. Thus, the text warns against using the global method for unreliable media and provides a method for implanting global information in the extended header for each file, rather than in the *typeflag* **g** records.

No facility for data translation or filtering on a per-file basis is included because the standard developers could not invent an interface that would allow this in an efficient manner. If a filter, such as encryption or compression, is to be applied to all the files, it is more efficient to apply the filter to the entire archive as a single file. The standard developers considered interfaces that would invoke a shell script for each file going into or out of the archive, but the system overhead in this approach was considered to be too high.

One such approach would be to have **filter=** records that give a pathname for an executable. When the program is invoked, the file and archive would be open for standard input/output and all the header fields would be available as environment variables or command-line arguments. The standard developers did discuss such schemes, but they were omitted from IEEE Std 1003.1-200x due to concerns about excessive overhead. Also, the program itself would need to be in the archive if it were to be used portably.

There is currently no portable means of identifying the character set(s) used for a file in the file system. Therefore, *pax* has not been given a mechanism to generate charset records automatically. The only portable means of doing this is for the user to write the archive using the

−ocharset=*string* command line option. This assumes that all of the files in the archive use the same encoding. The ''implementation-defined'' text is included to allow for a system that can identify the encodings used for each of its files.

The table of standards that accompanies the charset record description is acknowledged to be very limited. Only a limited number of character set standards is reasonable for maximal interchange. Any character set is, of course, possible by prior agreement. It was suggested that EBCDIC be listed, but it was omitted because it is not defined by a formal standard. Formal standards, and then only those with reasonably large followings, can be included here, simply as a matter of practicality. The <*value*>s represent names of officially registered character sets in the format required by the ISO 2375: 1985 standard.

The normal comma or <blank>-separated list rules are not followed in the case of keyword options to allow ease of argument parsing for *getopts*.

Further information on character encodings is in **pax Archive Character Set Encoding/Decoding** (on page 2931).

The standard developers have reserved keyword name space for vendor extensions. It is suggested that the format to be used is:

```
VENDOR.keyword
```

where *VENDOR* is the name of the vendor or organization in all uppercase letters. It is further suggested that the keyword following the period be named differently than any of the standard keywords so that it could be used for future standardization, if appropriate, by omitting the *VENDOR* prefix.

The <*length*> field in the extended header record was included to make it simpler to step through the records, even if a record contains an unknown format (to a particular *pax*) with complex interactions of special characters. It also provides a minor integrity checkpoint within the records to aid a program attempting to recover files from a damaged archive.

There are no extended header versions of the *devmajor* and *devminor* fields because the unspecified format **ustar** header field should be sufficient. If they are not, vendor-specific extended keywords (such as *VENDOR.devmajor*) should be used.

Device and *i*-number labeling of files was not adopted from *cpio*; files are interchanged strictly on a symbolic name basis, as in **ustar**.

Just as with the **ustar** format descriptions, the new format makes no special arrangements for multi-volume archives. Each of the *pax* archive types is assumed to be inside a single POSIX file and splitting that file over multiple volumes (diskettes, tape cartridges, and so on), processing their labels, and mounting each in the proper sequence are considered to be implementation details that cannot be described portably.

The *pax* format is intended for interchange, not only for backup on a single (family of) systems. It is not as densely packed as might be possible for backup:

- It contains information as coded characters that could be coded in binary.
- It identifies extended records with name fields that could be omitted in favor of a fixed-field layout.
- It translates names into a portable character set and identifies locale-related information, both of which are probably unnecessary for backup.

The requirements on restoring from an archive are slightly different from the historical wording, allowing for non-monolithic privilege to bring forward as much as possible. In particular, attributes such as ''high performance file'' might be broadly but not universally granted while set-user-ID or *chown*() might be much more restricted. There is no implication in IEEE Std 1003.1-200x that the security information be honored after it is restored to the file hierarchy, in spite of what might be improperly inferred by the silence on that topic. That is a topic for another standard.

Links are recorded in the fashion described here because a link can be to any file type. It is desirable in general to be able to restore part of an archive selectively and restore all of those files completely. If the data is not associated with each link, it is not possible to do this. However, the data associated with a file can be large, and when selective restoration is not needed, this can be a significant burden. The archive is structured so that files that have no associated data can always be restored by the name of any link name of any link, and the user may choose whether data is recorded with each instance of a file that contains data. The format permits mixing of both types of links in a single archive; this can be done for special needs, and *pax* is expected to interpret such archives on input properly, despite the fact that there is no *pax* option that would force this mixed case on output. (When **−o linkdata** is used, the output must contain the duplicate data, but the implementation is free to include it or omit it when **−o linkdata** is not used.)

The time values are included as extended header records for those implementations needing more than the eleven octal digits allowed by the **ustar** format. Portable file timestamps cannot be negative. If *pax* encounters a file with a negative timestamp in **copy** or **write** mode, it can reject the file, substitute a non-negative timestamp, or generate a non-portable timestamp with a leading `'-'`. Even though some implementations can support finer file-time granularities than seconds, the normative text requires support only for seconds since the Epoch because the ISO POSIX-1 standard states them that way. The **ustar** format includes only *mtime*; the new format adds *atime* and *ctime* for symmetry. The *atime* access time restored to the file system will be affected by the **−p a** and **−p e** options. The *ctime* creation time (actually *inode* modification time) is described with ‘‘appropriate privilege’’ so that it can be ignored when writing to the file system. POSIX does not provide a portable means to change file creation time. Nothing is intended to prevent a non-portable implementation of *pax* from restoring the value.

The *gid*, *size*, and *uid* extended header records were included to allow expansion beyond the sizes specified in the regular *tar* header. New file system architectures are emerging that will exhaust the 12-digit size field. There are probably not many systems requiring more than 8 digits for user and group IDs, but the extended header values were included for completeness, allowing overrides for all of the decimal values in the *tar* header.

The standard developers intended to describe the effective results of *pax* with regard to file ownerships and permissions; implementations are not restricted in timing or sequencing the restoration of such, provided the results are as specified.

Much of the text describing the extended headers refers to use in ‘‘**write** or **copy** modes’’. The **copy** mode references are due to the normative text: ‘‘The effect of the copy shall be as if the copied files were written to an archive file and then subsequently extracted ...’’. There is certainly no way to test whether *pax* is actually generating the extended headers in **copy** mode, but the effects must be as if it had.

pax Archive Character Set Encoding/Decoding

There is a need to exchange archives of files between systems of different native codesets. Filenames, group names, and user names must be preserved to the fullest extent possible when an archive is read on the receiving platform. Translation of the contents of files is not within the scope of the *pax* utility.

There will also be the need to represent characters that are not available on the receiving platform. These unsupported characters cannot be automatically folded to the local set of characters due to the chance of collisions. This could result in overwriting previous extracted files from the archive or pre-existing files on the system.

For these reasons, the codeset used to represent characters within the extended header records of the *pax* archive must be sufficiently rich to handle all commonly used character sets. The fields requiring translation include, at a minimum, filenames, user names, group names, and link pathnames. Implementations may wish to have localized extended keywords that use non-portable characters.

The standard developers considered the following options:

- The archive creator specifies the well-defined name of the source codeset. The receiver must then recognize the codeset name and perform the appropriate translations to the destination codeset.
- The archive creator includes within the archive the character mapping table for the source codeset used to encode extended header records. The receiver must then read the character mapping table and perform the appropriate translations to the destination codeset.
- The archive creator translates the extended header records in the source codeset into a canonical form. The receiver must then perform the appropriate translations to the destination codeset.

The approach that incorporates the name of the source codeset poses the problem of codeset name registration, and makes the archive useless to *pax* archive decoders that do not recognize that codeset.

Because parts of an archive may be corrupted, the standard developers felt that including the character map of the source codeset was too fragile. The loss of this one key component could result in making the entire archive useless. (The difference between this and the global extended header decision was that the latter has a workaround—duplicating extended header records on unreliable media—but this would be too burdensome for large character set maps.)

Both of the above approaches also put an undue burden on the *pax* archive receiver to handle the cross-product of all source and destination codesets.

To simplify the translation from the source codeset to the canonical form and from the canonical form to the destination codeset, the standard developers decided that the internal representation should be a stateless encoding. A stateless encoding is one where each codepoint has the same meaning, without regard to the decoder being in a specific state. An example of a stateful encoding would be the Japanese Shift-JIS; an example of a stateless encoding would be the ISO/IEC 646:1991 standard (equivalent to 7-bit ASCII).

For these reasons, the standard developers decided to adopt a canonical format for the representation of file information strings. The obvious, well-endorsed candidate is the ISO/IEC 10646-1:2000 standard (based in part on Unicode), which can be used to represent the characters of virtually all standardized character sets. The standard developers initially agreed upon using UCS2 (16-bit Unicode) as the internal representation. This repertoire of characters provides a sufficiently rich set to represent all commonly-used codesets.

However, the standard developers found that the 16-bit Unicode representation had some problems. It forced the issue of standardizing byte ordering. The 2-byte length of each character made the extended header records twice as long for the case of strings coded entirely from historical 7-bit ASCII. For these reasons, the standard developers chose the UTF-8 defined in the ISO/IEC 10646-1:2000 standard. This multi-byte representation encodes UCS2 or UCS4 characters reliably and deterministically, eliminating the need for a canonical byte ordering. In addition, NUL octets and other characters possibly confusing to POSIX file systems do not appear, except to represent themselves. It was realized that certain national codesets take up more space after the encoding, due to their placement within the UCS range; it was felt that the usefulness of the encoding of the names outweighs the disadvantage of size increase for file, user, and group names.

The encoding of UTF-8 is as follows:

```
UCS4 Hex Encoding  UTF-8 Binary Encoding

00000000-0000007F  0xxxxxxx
00000080-000007FF  110xxxxx 10xxxxxx
00000800-0000FFFF  1110xxxx 10xxxxxx 10xxxxxx
00010000-001FFFFF  11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000-03FFFFFF  111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000-7FFFFFFF  1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

where each `'x'` represents a bit value from the character being translated.

ustar Interchange Format

The description of the **ustar** format reflects numerous enhancements over pre-1988 versions of the historical *tar* utility. The goal of these changes was not only to provide the functional enhancements desired, but also to retain compatibility between new and old versions. This compatibility has been retained. Archives written using the old archive format are compatible with the new format.

Implementors should be aware that the previous file format did not include a mechanism to archive directory type files. For this reason, the convention of using a filename ending with slash was adopted to specify a directory on the archive.

The total size of the *name* and *prefix* fields have been set to meet the minimum requirements for {PATH_MAX}. If a pathname will fit within the *name* field, it is recommended that the pathname be stored there without the use of the *prefix* field. Although the name field is known to be too small to contain {PATH_MAX} characters, the value was not changed in this version of the archive file format to retain backward compatibility, and instead the prefix was introduced. Also, because of the earlier version of the format, there is no way to remove the restriction on the *linkname* field being limited in size to just that of the *name* field.

The *size* field is required to be meaningful in all implementation extensions, although it could be zero. This is required so that the data blocks can always be properly counted.

It is suggested that if device special files need to be represented that cannot be represented in the standard format that one of the extension types (**A-Z**) be used, and that the additional information for the special file be represented as data and be reflected in the *size* field.

Attempting to restore a special file type, where it is converted to ordinary data and conflicts with an existing filename, need not be specially detected by the utility. If run as an ordinary user, *pax* should not be able to overwrite the entries in, for example, **/dev** in any case (whether the file is converted to another type or not). If run as a privileged user, it should be able to do so, and it would be considered a bug if it did not. The same is true of ordinary data files and similarly named special files; it is impossible to anticipate the needs of the user (who could really intend to overwrite the file), so the behavior should be predictable (and thus regular) and rely on the protection system as required.

The value 7 in the *typeflag* field is intended to define how contiguous files can be stored in a **ustar** archive. IEEE Std 1003.1-200x does not require the contiguous file extension, but does define a standard way of archiving such files so that all conforming systems can interpret these file types in a meaningful and consistent manner. On a system that does not support extended file types, the *pax* utility should do the best it can with the file and go on to the next.

The file protection modes are those conventionally used by the *ls* utility. This is extended beyond the usage in the ISO POSIX-2 standard to support the ‘‘shared text’’ or ‘‘sticky’’ bit. It is intended that the conformance document should not document anything beyond the existence of and support of such a mode. Further extensions are expected to these bits, particularly with overloading the set-user-ID and set-group-ID flags.

cpio Interchange Format

The reference to appropriate privilege in the *cpio* format refers to an error on standard output; the **ustar** format does not make comparable statements.

The model for this format was the historical System V *cpio*–**c** data interchange format. This model documents the portable version of the *cpio* format and not the binary version. It has the flexibility to transfer data of any type described within IEEE Std 1003.1-200x, yet is extensible to transfer data types specific to extensions beyond IEEE Std 1003.1-200x (for example, contiguous files). Because it describes existing practice, there is no question of maintaining upward compatibility.

cpio Header

There has been some concern that the size of the *c_ino* field of the header is too small to handle those systems that have very large *inode* numbers. However, the *c_ino* field in the header is used strictly as a hard-link resolution mechanism for archives. It is not necessarily the same value as the *inode* number of the file in the location from which that file is extracted.

The name *c_magic* is based on historical usage.

cpio Filename

For most historical implementations of the *cpio* utility, {PATH_MAX} octets can be used to describe the pathname without the addition of any other header fields (the NUL character would be included in this count). {PATH_MAX} is the minimum value for pathname size, documented as 256 bytes. However, an implementation may use *c_namesize* to determine the exact length of the pathname. With the current description of the **<cpio.h>** header, this pathname size can be as large as a number that is described in six octal digits.

Two values are documented under the *c_mode* field values to provide for extensibility for known file types:

0110 000 Reserved for contiguous files. The implementation may treat the rest of the information for this archive like a regular file. If this file type is undefined, the implementation may create the file as a regular file.

This provides for extensibility of the *cpio* format while allowing for the ability to read old archives. Files of an unknown type may be read as ‘‘regular files’’ on some implementations. On a system that does not support extended file types, the *pax* utility should do the best it can with the file and go on to the next.

FUTURE DIRECTIONS

None.

SEE ALSO

cp, *ed*, *getopts*, *printf*, the Base Definitions volume of IEEE Std 1003.1-200x, **<cpio.h>**, the System Interfaces volume of IEEE Std 1003.1-200x, *chown*( ), *creat*( ), *mkdir*( ), *stat*( ), *write*( )

CHANGE HISTORY

First released in Issue 4.

Issue 5

A note is added to the APPLICATION USAGE indicating that the *cpio* and *tar* formats can only support files up to 8 gigabytes in size.

Issue 6

The *pax* utility is aligned with the IEEE P1003.2b draft standard:

- Support has been added for symbolic links in the options and interchange formats.
- A new format has been devised, based on extensions to ustar.
- References to the ''extended'' *tar* and *cpio* formats derived from the POSIX.1-1990 standard have been changed to remove the ''extended'' adjective because this could cause confusion with the extended *tar* header added in this revision. (All references to *tar* are actually to **ustar**).

IEEE PASC Interpretation 1003.2 #168 is applied clarifying that *mkdir*( ) and *mkfifo*( ) calls can ignore an [EEXIST] error when extracting an archive.

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

IEEE PASC Interpretation 1003.2 #180 is applied, clarifying how extracted files are created when in **read** mode.

IEEE PASC Interpretation 1003.2 #181 is applied, clarifying the description of the −**t** option. |

IEEE PASC Interpretation 1003.2 #195 is applied. |

IEEE PASC Interpretation 1003.2 #206 is applied, clarifying the handling of links for the −**H**, −**L**, and −**l** options. |

NAME

pr — print files

SYNOPSIS

```
pr [+page][−column][−adFmrt][−e[char][gap]][−h header][−i[char][gap]]
    [−l lines][−n[char][width]][−o offset][−s[char]][−w width][−fp]
    [file...]
```

XSI

DESCRIPTION

The *pr* utility is a printing and pagination filter. If multiple input files are specified, each shall be read, formatted, and written to standard output. By default, the input shall be separated into 66-line pages, each with:

- A 5-line header that includes the page number, date, time, and the pathname of the file
- A 5-line trailer consisting of blank lines

If standard output is associated with a terminal, diagnostic messages shall be deferred until the *pr* utility has completed processing.

When options specifying multi-column output are specified, output text columns shall be of equal width; input lines that do not fit into a text column shall be truncated. By default, text columns shall be separated with at least one <blank>.

OPTIONS

The *pr* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that: the *page* option has a `'+'` delimiter; *page* and *column* can be multi-digit numbers; some of the option-arguments are optional; and some of the option-arguments cannot be specified as separate arguments from the preceding option letter. In particular, the **−s** option does not allow the option letter to be separated from its argument, and the options **−e**, **−i**, and **−n** require that both arguments, if present, not be separated from the option letter.

The following options shall be supported. In the following option descriptions, *column*, *lines*, *offset*, *page*, and *width* are positive decimal integers; *gap* is a non-negative decimal integer.

+*page*	Begin output at page number *page* of the formatted input.
−*column*	Produce multi-column output that is arranged in *column* columns (the default shall be 1) and is written down each column in the order in which the text is received from the input file. This option should not be used with **−m**. The options **−e** and **−i** shall be assumed for multiple text-column output. Whether or not text columns are produced with identical vertical lengths is unspecified, but a text column shall never exceed the length of the page (see the **−l** option). When used with **−t**, use the minimum number of lines to write the output.
−a	Modify the effect of the **−***column* option so that the columns are filled across the page in a round-robin order (for example, when *column* is 2, the first input line heads column 1, the second heads column 2, the third is the second line in column 1, and so on).
−d	Produce output that is double-spaced; append an extra <newline> following every <newline> found in the input.
−e[*char***][***gap***]**	Expand each input <tab> to the next greater column position specified by the formula $n*gap+1$, where *n* is an integer > 0. If *gap* is zero or is omitted, it shall default to 8. All <tab>s in the input shall be expanded into the appropriate number of <space>s. If any non-digit character, *char*, is specified, it shall be used as the

input <tab>.

XSI **−f**	Use a <form-feed> for new pages, instead of the default behavior that uses a sequence of <newline>s. Pause before beginning the first page if the standard output is associated with a terminal.
−F	Use a <form-feed> for new pages, instead of the default behavior that uses a sequence of <newline>s.
−h *header*	Use the string *header* to replace the contents of the *file* operand in the page header.
−i[*char***][***gap***]**	In output, replace multiple <space>s with <tab>s wherever two or more adjacent <space>s reach column positions *gap*+1, 2* *gap*+1, 3* *gap*+1, and so on. If *gap* is zero or is omitted, default tab settings at every eighth column position shall be assumed. If any non-digit character, *char*, is specified, it shall be used as the output <tab>.
−l *lines*	Override the 66-line default and reset the page length to *lines*. If *lines* is not greater than the sum of both the header and trailer depths (in lines), the *pr* utility shall suppress both the header and trailer, as if the **−t** option were in effect.
−m	Merge files. Standard output shall be formatted so the *pr* utility writes one line from each file specified by a *file* operand, side by side into text columns of equal fixed widths, in terms of the number of column positions. Implementations shall support merging of at least nine *file* operands.
−n[*char***][***width***]**	Provide *width*-digit line numbering (default for *width* shall be 5). The number shall occupy the first *width* column positions of each text column of default output or each line of **−m** output. If *char* (any non-digit character) is given, it shall be appended to the line number to separate it from whatever follows (default for *char* is a <tab>).
−o *offset*	Each line of output shall be preceded by offset <space>s. If the **−o** option is not specified, the default offset shall be zero. The space taken is in addition to the output line width (see the **−w** option below).
−p	Pause before beginning each page if the standard output is directed to a terminal (*pr* shall write an <alert> to standard error and wait for a <carriage-return> to be read on **/dev/tty**).
−r	Write no diagnostic reports on failure to open files.
−s[*char***]**	Separate text columns by the single character *char* instead of by the appropriate number of <space>s (default for *char* shall be <tab>).
−t	Write neither the five-line identifying header nor the five-line trailer usually supplied for each page. Quit writing after the last line of each file without spacing to the end of the page.
−w *width*	Set the width of the line to *width* column positions for multiple text-column output only. If the **−w** option is not specified and the **−s** option is not specified, the default width shall be 72. If the **−w** option is not specified and the **−s** option is specified, the default width shall be 512. For single column output, input lines shall not be truncated.

OPERANDS

The following operand shall be supported:

file	A pathname of a file to be written. If no *file* operands are specified, or if a *file* operand is `'−'`, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'−'`. See the INPUT FILES section.

INPUT FILES

The input files shall be text files.

The file **/dev/tty** shall be used to read responses required by the −**p** option.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *pr*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and which characters are defined as printable (character class **print**). Non-printable characters are still written to standard output, but are not counted for the purpose for column-width and line-length calculations.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_TIME	Determine the format of the date and time for use in writing header lines.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
TZ	Determine the timezone used to calculate date and time strings written in header lines. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

If *pr* receives an interrupt while writing to a terminal, it shall flush all accumulated error messages to the screen before terminating.

STDOUT

The *pr* utility output shall be a paginated version of the original file (or files). This pagination shall be accomplished using either <form-feed>s or a sequence of <newline>s, as controlled by XSI the −**F** or −**f** option. Page headers shall be generated unless the −**t** option is specified. The page headers shall be of the form:

```
"\n\n%s %s Page %d\n\n\n", <output of date>, <file>, <page number>
```

In the POSIX locale, the <*output of date*> field, representing the date and time of last modification of the input file (or the current date and time if the input file is standard input), shall be equivalent to the output of the following command as it would appear if executed at the given time:

```
date "+%b %e %H:%M %Y"
```

without the trailing <newline>, if the page being written is from standard input. If the page being written is not from standard input, in the POSIX locale, the same format shall be used, but the time used shall be the modification time of the file corresponding to *file* instead of the current time. When the *LC_TIME* locale category is not set to the POSIX locale, a different format and order of presentation of this field may be used.

If the standard input is used instead of a *file* operand, the <*file*> field shall be replaced by a null string.

If the −**h** option is specified, the <*file*> field shall be replaced by the *header* argument.

STDERR

The standard error shall be used for diagnostic messages and for alerting the terminal when −**p** is specified. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. Print a numbered list of all files in the current directory:

   ```
   ls −a | pr −n −h "Files in $(pwd)."
   ```

2. Print **file1** and **file2** as a double-spaced, three-column listing headed by ‘‘file list’’:

   ```
   pr −3d −h "file list" file1 file2
   ```

3. Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, . . .:

   ```
   pr −e9 −t <file1 >file2
   ```

RATIONALE

This utility is one of those that does not follow the Utility Syntax Guidelines because of its historical origins. The standard developers could have added new options that obeyed the guidelines (and marked the old options *obsolescent*) or devised an entirely new utility; there are examples of both actions in this volume of IEEE Std 1003.1-200x. Because of its widespread use by historical applications, the standard developers decided to exempt this version of *pr* from many of the guidelines.

Implementations are required to accept option-arguments to the −**h**, −**l**, −**o**, and −**w** options whether presented as part of the same argument or as a separate argument to *pr*, as suggested by the Utility Syntax Guidelines. The −**n** and −**s** options, however, are specified as in historical practice because they are frequently specified without their optional arguments. If a <blank>

were allowed before the option-argument in these cases, a *file* operand could mistakenly be interpreted as an option-argument in historical applications.

The text about the minimum number of lines in multi-column output was included to ensure that a best effort is made in balancing the length of the columns. There are known historical implementations in which, for example, 60-line files are listed by *pr* –2 as one column of 56 lines and a second of 4. Although this is not a problem when a full page with headers and trailers is produced, it would be relatively useless when used with **–t**.

Historical implementations of the *pr* utility have differed in the action taken for the **–f** option. BSD uses it as described here for the **–F** option; System V uses it to change trailing <newline>s on each page to a <form-feed> and, if standard output is a TTY device, sends an <alert> to standard error and reads a line from **/dev/tty** before the first page. There were strong arguments from both sides of this issue concerning historical practice and as a result the **–F** option was added. XSI-conformant systems support the System V historical actions for the **–f** option.

The <*output of date*> field in the **–l** format is specified only for the POSIX locale. As noted, the format can be different in other locales. No mechanism for defining this is present in this volume of IEEE Std 1003.1-200x, as the appropriate vehicle is a message catalog; that is, the format should be specified as a ‘‘message’’.

FUTURE DIRECTIONS

None.

SEE ALSO

expand, *lp*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **–p** option is added.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

printf — write formatted output

SYNOPSIS

```
printf format[argument...]
```

DESCRIPTION

The *printf* utility shall write formatted operands to the standard output. The *argument* operands shall be formatted under control of the *format* operand.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

format	A string describing the format to use to write the remaining operands. See the EXTENDED DESCRIPTION section.
argument	The strings to be written to standard output, under the control of *format*. See the EXTENDED DESCRIPTION section.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *printf*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_NUMERIC	Determine the locale for numeric formatting. It shall affect the format of numbers written using the `e`, `E`, `f`, `g`, and `G` conversion specifier characters (if supported).
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the EXTENDED DESCRIPTION section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The *format* operand shall be used as the *format* string described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation with the following exceptions:

1. A <space> in the format string, in any context other than a flag of a conversion specification, shall be treated as an ordinary character that is copied to the output.
2. A `'Δ'` character in the format string shall be treated as a `'Δ'` character, not as a <space>.
3. In addition to the escape sequences shown in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`), `"\ddd"`, where *ddd* is a one, two, or three-digit octal number, shall be written as a byte with the numeric value specified by the octal number.
4. The implementation shall not precede or follow output from the `d` or `u` conversion specifiers with <blank>s not specified by the *format* operand.
5. The implementation shall not precede output from the `o` conversion specifier with zeros not specified by the *format* operand.
6. The `e`, `E`, `f`, `g`, and `G` conversion specifiers need not be supported.
7. An additional conversion specifier character, `b`, shall be supported as follows. The argument shall be taken to be a string that may contain backslash-escape sequences. The following backslash-escape sequences shall be supported:

   — The escape sequences listed in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`), which shall be converted to the characters they represent

   — `"\0ddd"`, where *ddd* is a zero, one, two, or three-digit octal number that shall be converted to a byte with the numeric value specified by the octal number

   — `'\c'`, which shall not be written and shall cause *printf* to ignore any remaining characters in the string operand containing it, any remaining string operands, and any additional characters in the *format* operand

   The interpretation of a backslash followed by any other sequence of characters is unspecified.

   Bytes from the converted string shall be written until the end of the string or the number of bytes indicated by the precision specification is reached. If the precision is omitted, it shall be taken to be infinite, so all bytes up to the end of the converted string shall be written.
8. For each conversion specification that consumes an argument, the next argument operand shall be evaluated and converted to the appropriate type for the conversion as specified below.
9. The *format* operand shall be reused as often as necessary to satisfy the argument operands. Any extra `c` or `s` conversion specifiers shall be evaluated as if a null string argument were supplied; other extra conversion specifications shall be evaluated as if a zero argument were supplied. If the *format* operand contains no conversion specifications and *argument* operands are present, the results are unspecified.

10. If a character sequence in the *format* operand begins with a `'%'` character, but does not form a valid conversion specification, the behavior is unspecified.

The *argument* operands shall be treated as strings if the corresponding conversion specifier is `b`, `c`, or `s`; otherwise, it shall be evaluated as a C constant, as described by the ISO C standard, with the following extensions:

- A leading plus or minus sign shall be allowed.
- If the leading character is a single-quote or double-quote, the value shall be the numeric value in the underlying codeset of the character following the single-quote or double-quote.

If an argument operand cannot be completely converted into an internal value appropriate to the corresponding conversion specification, a diagnostic message shall be written to standard error and the utility shall not exit with a zero exit status, but shall continue processing any remaining operands and shall write the value accumulated at the time the error was detected to standard output.

It is not considered an error if an argument operand is not completely used for a `c` or `s` conversion or if a string operand's first or second character is used to get the numeric value of a character.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The floating-point formatting conversion specifications of *printf*() are not required because all arithmetic in the shell is integer arithmetic. The *awk* utility performs floating-point calculations and provides its own **printf** function. The *bc* utility can perform arbitrary-precision floating-point arithmetic, but does not provide extensive formatting capabilities. (This *printf* utility cannot really be used to format *bc* output; it does not support arbitrary precision.) Implementations are encouraged to support the floating-point conversions as an extension.

Note that this *printf* utility, like the *printf*() function defined in the System Interfaces volume of IEEE Std 1003.1-200x on which it is based, makes no special provision for dealing with multi-byte characters when using the `%c` conversion specification or when a precision is specified in a `%b` or `%s` conversion specification. Applications should be extremely cautious using either of these features when there are multi-byte characters in the character set.

No provision is made in this volume of IEEE Std 1003.1-200x which allows field widths and precisions to be specified as `'*'` since the `'*'` can be replaced directly in the *format* operand using shell variable substitution. Implementations can also provide this feature as an extension if they so choose.

Hexadecimal character constants as defined in the ISO C standard are not recognized in the *format* operand because there is no consistent way to detect the end of the constant. Octal character constants are limited to, at most, three octal digits, but hexadecimal character constants are only terminated by a non-hex-digit character. In the ISO C standard, the `"##"` concatenation operator can be used to terminate a constant and follow it with a hexadecimal character to be written. In the shell, concatenation occurs before the *printf* utility has a chance to parse the end of the hexadecimal constant.

The `%b` conversion specification is not part of the ISO C standard; it has been added here as a portable way to process backslash escapes expanded in string operands as provided by the *echo* utility. See also the APPLICATION USAGE section of *echo* (on page 2534) for ways to use *printf* as a replacement for all of the traditional versions of the *echo* utility.

If an argument cannot be parsed correctly for the corresponding conversion specification, the *printf* utility is required to report an error. Thus, overflow and extraneous characters at the end of an argument being used for a numeric conversion shall be reported as errors.

EXAMPLES

To alert the user and then print and read a series of prompts:

```
printf "\aPlease fill in the following: \nName: "
read name
printf "Phone number: "
read phone
```

To read out a list of right and wrong answers from a file, calculate the percentage correctly, and print them out. The numbers are right-justified and separated by a single <tab>. The percentage is written to one decimal place of accuracy:

```
while read right wrong ; do
    percent=$(echo "scale=1;($right*100)/($right+$wrong)" | bc)
    printf "%2d right\t%2d wrong\t(%s%%)\n" \
        $right $wrong $percent
done < database_file
```

The command:

```
printf "%5d%4d\n" 1 21 321 4321 54321
```

produces:

```
    1  21
  3214321
54321   0
```

Note that the *format* operand is used three times to print all of the given strings and that a `'0'` was supplied by *printf* to satisfy the last `%4d` conversion specification.

The *printf* utility is required to notify the user when conversion errors are detected while producing numeric output; thus, the following results would be expected on an implementation with 32-bit twos-complement integers when `%d` is specified as the *format* operand:

Argument	Standard Output	Diagnostic Output
5a	5	**printf: "5a" not completely converted**
9999999999	2147483647	**printf: "9999999999" arithmetic overflow**
−9999999999	−2147483648	**printf: "−9999999999" arithmetic overflow**
ABC	0	**printf: "ABC" expected numeric value**

The diagnostic message format is not specified, but these examples convey the type of information that should be reported. Note that the value shown on standard output is what would be expected as the return value from the *strtol*( ) function as defined in the System Interfaces volume of IEEE Std 1003.1-200x. A similar correspondence exists between `%u` and *strtoul*( ) and `%e`, `%f`, and `%g` (if the implementation supports floating-point conversions) and *strtod*( ).

In a locale using the ISO/IEC 646:1991 standard as the underlying codeset, the command:

```
printf "%d\n" 3 +3 -3 \'3 \"+3 "'-3"
```

produces:

3	Numeric value of constant 3
3	Numeric value of constant 3
−3	Numeric value of constant −3
51	Numeric value of the character `'3'` in the ISO/IEC 646:1991 standard codeset
43	Numeric value of the character `'+'` in the ISO/IEC 646:1991 standard codeset
45	Numeric value of the character `'-'` in the ISO/IEC 646:1991 standard codeset

Note that in a locale with multi-byte characters, the value of a character is intended to be the value of the equivalent of the **wchar_t** representation of the character as described in the System Interfaces volume of IEEE Std 1003.1-200x.

RATIONALE

The *printf* utility was added to provide functionality that has historically been provided by *echo*. However, due to irreconcilable differences in the various versions of *echo* extant, the version has few special features, leaving those to this new *printf* utility, which is based on one in the Ninth Edition system.

The EXTENDED DESCRIPTION section almost exactly matches the *printf*( ) function in the ISO C standard, although it is described in terms of the file format notation in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 5, File Format Notation.

FUTURE DIRECTIONS

None.

SEE ALSO

awk, *bc*, *echo*, the System Interfaces volume of IEEE Std 1003.1-200x, *printf*( )

CHANGE HISTORY

First released in Issue 4.

NAME

prs — print an SCCS file (**DEVELOPMENT**)

SYNOPSIS

XSI
```
prs [−a][−d dataspec][−r[SID]] file...
prs [ −e| −l] −c cutoff [−d dataspec] file...
prs [ −e| −l] −r[SID][−d dataspec]file...
```

DESCRIPTION

The *prs* utility shall write to standard output parts or all of an SCCS file in a user-supplied format.

OPTIONS

The *prs* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the **−r** option has an optional option-argument. This optional option-argument cannot be presented as a separate argument. The following options shall be supported:

−d *dataspec*	Specify the output data specification. The *dataspec* shall be a string consisting of SCCS file *data keywords* (see **Data Keywords** (on page 2946)) interspersed with optional user-supplied text.
−r[*SID***]**	Specify the SCCS identification string (SID) of a delta for which information is desired. If no *SID* option-argument is specified, the SID of the most recently created delta shall be assumed.
−e	Request information for all deltas created earlier than and including the delta designated via the **−r** option or the date-time given by the **−c** option.
−l	Request information for all deltas created later than and including the delta designated via the **−r** option or the date-time given by the **−c** option.
−c *cutoff*	Indicate the *cutoff* date-time, in the form: `YY[MM[DD[HH[MM[SS]]]]]` For the *YY* component, values in the range [69,99] shall refer to years 1969 to 1999 inclusive, and values in the range [00,68] shall refer to years 2000 to 2068 inclusive. **Note:** It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.) No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time shall be included in the output. Units omitted from the date-time default to their maximum possible values; for example, **−c 7502** is equivalent to **−c 750228235959**.
−a	Request writing of information for both removed, that is, *delta type*=*R* (see *rmdel* (on page 3027)) and existing, that is, *delta type*=*D*, deltas. If the **−a** option is not specified, information for existing deltas only shall be provided.

OPERANDS

The following operand shall be supported:

file	A pathname of an existing SCCS file or a directory. If *file* is a directory, the *prs* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin

with **s.**) and unreadable files shall be silently ignored.

If exactly one *file* operand appears, and it is `'−'`, the standard input shall be read; each line of the standard input shall be taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only when the *file* operand is specified as `'−'`. Each line of the text file shall be interpreted as an SCCS pathname.

INPUT FILES

Any SCCS files displayed are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *prs*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be a text file whose format is dependent on the data keywords specified with the −**d** option.

Data Keywords

Data keywords specify which parts of an SCCS file shall be retrieved and output. All parts of an SCCS file have an associated data keyword. A data keyword may appear in a *dataspec* multiple times.

The information written by *prs* shall consist of:

1. The user-supplied text
2. Appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*

The format of a data keyword value shall either be simple (`'S'`), in which keyword substitution is direct, or multi-line (`'M'`).

User-supplied text shall be any text other than recognized data keywords. A <tab> shall be specified by `'\t'` and <newline> by `'\n'`. When the −**r** option is not specified, the default *dataspec* shall be:

```
:PN::\n\n
```

and the following *dataspec* shall be used for each selected delta:

```
:Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:
```

SCCS File Data Keywords

Keyword	Data Item	File Section	Value	Format
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	**:Li:/:Ld:/:Lu:**	S
:Li:	Lines inserted by Delta	"	_nnnnn_***	S
:Ld:	Lines deleted by Delta	"	_nnnnn_***	S
:Lu:	Lines unchanged by Delta	"	_nnnnn_***	S
:DT:	Delta type	"	**D** or **R**	S
:I:	SCCS ID string (SID)	"	See below**	S
:R:	Release number	"	_nnnn_	S
:L:	Level number	"	_nnnn_	S
:B:	Branch number	"	_nnnn_	S
:S:	Sequence number	"	_nnnn_	S
:D:	Date delta created	"	**:Dy:/:Dm:/:Dd:**	S
:Dy:	Year delta created	"	_nn_	S
:Dm:	Month delta created	"	_nn_	S
:Dd:	Day delta created	"	_nn_	S
:T:	Time delta created	"	**:Th:::Tm:::Ts:**	S
:Th:	Hour delta created	"	_nn_	S
:Tm:	Minutes delta created	"	_nn_	S
:Ts:	Seconds delta created	"	_nn_	S
:P:	Programmer who created Delta	"	_logname_	S
:DS:	Delta sequence number	"	_nnnn_	S
:DP:	Predecessor Delta sequence number	"	_nnnn_	S
:DI:	Sequence number of deltas included, excluded or ignored	"	**:Dn:/:Dx:/:Dg:**	S
:Dn:	Deltas included (sequence #)	"	**:DS: :DS: . . .**	S
:Dx:	Deltas excluded (sequence #)	"	**:DS: :DS: . . .**	S
:Dg:	Deltas ignored (sequence #)	"	**:DS: :DS: . . .**	S
:MR:	MR numbers for delta	"	_text_	M
:C:	Comments for delta	"	_text_	M
:UN:	User names	User Names	_text_	M
:FL:	Flag list	Flags	_text_	M
:Y:	Module type flag	"	_text_	S
:MF:	MR validation flag	"	**yes** or **no**	S
:MP:	MR validation program name	"	_text_	S
:KF:	Keyword error, warning flag	"	**yes** or **no**	S
:KV:	Keyword validation string	"	_text_	S
:BF:	Branch flag	"	**yes** or **no**	S
:J:	Joint edit flag	"	**yes** or **no**	S
:LK:	Locked releases	"	**:R: . . .**	S
:Q:	User-defined keyword	"	_text_	S

SCCS File Data Keywords				
Keyword	**Data Item**	**File Section**	**Value**	**Format**
:M:	Module name	"	*text*	S
:FB:	Floor boundary	"	**:R:**	S
:CB:	Ceiling boundary	"	**:R:**	S
:Ds:	Default SID	"	**:I:**	S
:ND:	Null delta flag	"	**yes** or **no**	S
:FD:	File descriptive text	Comments	*text*	M
:BD:	Body	Body	*text*	M
:GB:	Gotten body	"	*text*	M
:W:	A form of *what* string	N/A	**:Z::M:\t:I:**	S
:A:	A form of *what* string	N/A	**:Z::Y: :M: :I::Z:**	S
:Z:	*what* string delimiter	N/A	@(#)	S
:F:	SCCS filename	N/A	*text*	S
:PN:	SCCS file pathname	N/A	*text*	S

* **:Dt:=:DT: :I: :D: :T: :P: :DS: :DP:**

** **:R:.:L:.:B:.:S:** if the delta is a branch delta (**:BF:**= =**yes**)
:R:.:L: if the delta is not a branch delta (**:BF:**= =**no**)

*** The line statistics are capped at 99 999. For example, if 100 000 lines were unchanged in a certain revision, **:Lu:** shall produce the value 99 999.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. The following example:

```
prs −d "User Names for :F: are:\n:UN:" s.file
```

might write to standard output:

```
User Names for s.file are:
xyz
131
abc
```

2. The following example:

```
prs −d "Delta for pgm :M:: :I: − :D: By :P:" −r s.file
```

might write to standard output:

```
Delta for pgm main.c: 3.7 − 77/12/01 By cas
```

3. As a special case:

```
prs s.file
```

might write to standard output:

```
s.file:
<blank line>
D 1.1 77/12/01 00:00:00 cas 1 000000/00000/00000
MRs:
bl78−12345
bl79−54321
COMMENTS:
this is the comment line for s.file initial delta
<blank line>
```

for each delta table entry of the **D** type. The only option allowed to be used with this special case is the −**a** option.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

admin, *delta*, *get*, *what*

CHANGE HISTORY

First released in Issue 2.

Issue 5

The phrase ''in which keyword substitution is followed by a <newline>'' is deleted from the end of the second paragraph of **Data Keywords** (on page 2946).

The interpretation of the *YY* component of the −**c** *cutoff* argument is noted.

Issue 6

The normative text is reworded to emphasize the term ''shall'' for implementation requirements.

The Open Group Base Resolution bwg2001-007 is applied, updating the table in STDOUT with a note that line statistics are capped at 99 999 for the **:Li:**, **:Ld:**, **:Lu:**, and **:DL:** keywords.

The Open Group Interpretation PIN4C.00009 is applied.

NAME

ps — report process status

SYNOPSIS

UP XSI

```
ps [−aA][−defl][−G grouplist][−o format]...[−p proclist][−t termlist]
   [−U userlist][−g grouplist][−n namelist][−u userlist]
```

DESCRIPTION

The *ps* utility shall write information about processes, subject to having the appropriate privileges to obtain information about those processes.

By default, *ps* shall select all processes with the same effective user ID as the current user and the same controlling terminal as the invoker.

OPTIONS

The *ps* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

	−a	Write information for all processes associated with terminals. Implementations may omit session leaders from this list.
	−A	Write information for all processes.
XSI	**−d**	Write information for all processes, except session leaders.
XSI	**−e**	Write information for all processes. (Equivalent to −**A**.)
XSI	**−f**	Generate a **full** listing. (See the STDOUT section for the contents of a **full** listing.)
XSI	**−g** *grouplist*	Write information for processes whose session leaders are given in *grouplist*. The application shall ensure that the *grouplist* is a single argument in the form of a <blank> or comma-separated list.
	−G *grouplist*	Write information for processes whose real group ID numbers are given in *grouplist*. The application shall ensure that the *grouplist* is a single argument in the form of a <blank> or comma-separated list.
XSI	**−l**	Generate a **long** listing. (See STDOUT for the contents of a **long** listing.)
XSI	**−n** *namelist*	Specify the name of an alternative system *namelist* file in place of the default. The name of the default file and the format of a *namelist* file are unspecified.
	−o *format*	Write information according to the format specification given in *format*. This is fully described in the STDOUT section. Multiple −**o** options can be specified; the format specification shall be interpreted as the <space>-separated concatenation of all the *format* option-arguments.
	−p *proclist*	Write information for processes whose process ID numbers are given in *proclist*. The application shall ensure that the *proclist* is a single argument in the form of a <blank> or comma-separated list.
XSI	**−t** *termlist*	Write information for processes associated with terminals given in *termlist*. The application shall ensure that the *termlist* is a single argument in the form of a <blank> or comma-separated list. Terminal identifiers shall be given in an implementation-defined format. On XSI-conformant systems, they shall be given in one of two forms: the device's filename (for example, **tty04**) or, if the device's filename starts with **tty**, just the identifier following the characters **tty** (for

example, `"04"`).

XSI **−u** *userlist*	Write information for processes whose user ID numbers or login names are given in *userlist*. The application shall ensure that the *userlist* is a single argument in the form of a <blank> or comma-separated list. In the listing, the numerical user ID shall be written unless the **−f** option is used, in which case the login name shall be written.
−U *userlist*	Write information for processes whose real user ID numbers or login names are given in *userlist*. The application shall ensure that the *userlist* is a single argument in the form of a <blank> or comma-separated list.

With the exception of **−o** *format*, all of the options shown are used to select processes. If any are specified, the default list shall be ignored and *ps* shall select the processes represented by the inclusive OR of all the selection-criteria options.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ps*:

COLUMNS	Override the system-selected horizontal display line size, used to determine the number of text columns to display. See the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables for valid values and results when it is unset or null.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
LC_TIME	Determine the format and contents of the date and time strings displayed.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
TZ	Determine the timezone used to calculate date and time strings displayed. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the −**o** option is not specified, the standard output format is unspecified.

XSI On XSI-conformant systems, the output format shall be as follows. The column headings and descriptions of the columns in a *ps* listing are given below. The precise meanings of these fields are implementation-defined. The letters `'f'` and `'l'` (below) indicate the option (**full** or **long**) that shall cause the corresponding heading to appear; **all** means that the heading always appears. Note that these two options determine only what information is provided for a process; they do not determine which processes are listed.

F	(l)	Flags (octal and additive) associated with the process.
S	(l)	The state of the process.
UID	(f,l)	The user ID number of the process owner; the login name is printed under the −**f** option.
PID	(all)	The process ID of the process; it is possible to kill a process if this datum is known.
PPID	(f,l)	The process ID of the parent process.
C	(f,l)	Processor utilization for scheduling.
PRI	(l)	The priority of the process; higher numbers mean lower priority.
NI	(l)	Nice value; used in priority computation.
ADDR	(l)	The address of the process.
SZ	(l)	The size in blocks of the core image of the process.
WCHAN	(l)	The event for which the process is waiting or sleeping; if blank, the process is running.
STIME	(f)	Starting time of the process.
TTY	(all)	The controlling terminal for the process.
TIME	(all)	The cumulative execution time for the process.
CMD	(all)	The command name; the full command name and its arguments are written under the −**f** option.

A process that has exited and has a parent, but has not yet been waited for by the parent, shall be marked **defunct**.

Under the option −**f**, *ps* tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the option −**f**, is written in square brackets.

The −**o** option allows the output format to be specified under user control.

The application shall ensure that the format specification is a list of names presented as a single argument, <blank> or comma-separated. Each variable has a default header. The default header can be overridden by appending an equals sign and the new text of the header. The rest of the characters in the argument shall be used as the header text. The fields specified shall be written in the order specified on the command line, and should be arranged in columns in the output. The field widths shall be selected by the system to be at least as wide as the header text (default or overridden value). If the header text is null, such as −**o** *user*=, the field width shall be at least as wide as the default header text. If all header text fields are null, no header line shall be written.

The following names are recognized in the POSIX locale:

ruser	The real user ID of the process. This shall be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise.
user	The effective user ID of the process. This shall be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise.
rgroup	The real group ID of the process. This shall be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise.
group	The effective group ID of the process. This shall be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise.
pid	The decimal value of the process ID.
ppid	The decimal value of the parent process ID.
pgid	The decimal value of the process group ID.
pcpu	The ratio of CPU time used recently to CPU time available in the same period, expressed as a percentage. The meaning of ‘‘recently’’ in this context is unspecified. The CPU time available is determined in an unspecified manner.
vsz	The size of the process in (virtual) memory in 1 024 byte units as a decimal integer.
nice	The decimal value of the nice value of the process; see *nice* (on page 2863).
etime	In the POSIX locale, the elapsed time since the process was started, in the form: `[[dd–]hh:]mm:ss` where *dd* shall represent the number of days, *hh* the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The *dd* field shall be a decimal integer. The *hh*, *mm*, and *ss* fields shall be two-digit decimal integers padded on the left with zeros.
time	In the POSIX locale, the cumulative CPU time of the process in the form: `[dd–]hh:mm:ss` The *dd*, *hh*, *mm*, and *ss* fields shall be as described in the **etime** specifier.
tty	The name of the controlling terminal of the process (if any) in the same format used by the *who* utility.
comm	The name of the command being executed (*argv*[0] value) as a string.
args	The command with all its arguments as a string. The implementation may truncate this value to the field width; it is implementation-defined whether any further truncation occurs. It is unspecified whether the string represented is a version of the argument list as it was passed to the command when it started, or is a version of the arguments as they may have been modified by the application. Applications cannot depend on being able to modify their argument list and having that modification be reflected in the output of *ps*.

Any field need not be meaningful in all implementations. In such a case a hyphen (`'−'`) should be output in place of the field value.

Only **comm** and **args** shall be allowed to contain <blank>s; all others shall not. Any implementation-defined variables shall be specified in the system documentation along with the default header and indicating if the field may contain <blank>s.

The following table specifies the default header to be used in the POSIX locale corresponding to each format specifier.

Table 4-17 Variable Names and Default Headers in *ps*

Format Specifier	Default Header	Format Specifier	Default Header
args	**COMMAND**	**ppid**	**PPID**
comm	**COMMAND**	**rgroup**	**RGROUP**
etime	**ELAPSED**	**ruser**	**RUSER**
group	**GROUP**	**time**	**TIME**
nice	**NI**	**tty**	**TT**
pcpu	**%CPU**	**user**	**USER**
pgid	**PGID**	**vsz**	**VSZ**
pid	**PID**		

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Things can change while *ps* is running; the snapshot it gives is only true for an instant, and might not be accurate by the time it is displayed.

The **args** format specifier is allowed to produce a truncated version of the command arguments. In some implementations, this information is no longer available when the *ps* utility is executed.

If the field width is too narrow to display a textual ID, the system may use a numeric version. Normally, the system would be expected to choose large enough field widths, but if a large number of fields were selected to write, it might squeeze fields to their minimum sizes to fit on one line. One way to ensure adequate width for the textual IDs is to override the default header for a field to make it larger than most or all user or group names.

There is no special quoting mechanism for header text. The header text is the rest of the argument. If multiple header changes are needed, multiple −**o** options can be used, such as:

```
ps −o "user=User Name" −o pid=Process\ ID
```

On some implementations, especially multi-level secure systems, *ps* may be severely restricted and produce information only about child processes owned by the user.

EXAMPLES

The command:

```
ps −o user,pid,ppid=MOM −o args
```

writes at least the following in the POSIX locale:

```
  USER   PID   MOM   COMMAND
helene    34    12   ps −o uid,pid,ppid=MOM −o args
```

The contents of the **COMMAND** field need not be the same in all implementations, due to possible truncation.

RATIONALE

There is very little commonality between BSD and System V implementations of *ps*. Many options conflict or have subtly different usages. The standard developers attempted to select a set of options for the base standard that were useful on a wide range of systems and selected options that either can be implemented on both BSD and System V-based systems without breaking the current implementations or where the options are sufficiently similar that any changes would not be unduly problematic for users or implementors.

It is recognized that on some implementations, especially multi-level secure systems, *ps* may be nearly useless. The default output has therefore been chosen such that it does not break historical implementations and also is likely to provide at least some useful information on most systems.

The major change is the addition of the format specification capability. The motivation for this invention is to provide a mechanism for users to access a wider range of system information, if the system permits it, in a portable manner. The fields chosen to appear in this volume of IEEE Std 1003.1-200x were arrived at after considering what concepts were likely to be both reasonably useful to the ‘‘average’’ user and had a reasonable chance of being implemented on a wide range of systems. Again it is recognized that not all systems are able to provide all the information and, conversely, some may wish to provide more. It is hoped that the approach adopted will be sufficiently flexible and extensible to accommodate most systems. Implementations may be expected to introduce new format specifiers.

The default output should consist of a short listing containing the process ID, terminal name, cumulative execution time, and command name of each process.

The preference of the standard developers would have been to make the format specification an operand of the *ps* command. Unfortunately, BSD usage precluded this.

At one time a format was included to display the environment array of the process. This was deleted because there is no portable way to display it.

The −**A** option is equivalent to the BSD −**g** and the SVID −**e**. Because the two systems differed, a mnemonic compromise was selected.

The −**a** option is described with some optional behavior because the SVID omits session leaders, but BSD does not.

In an early proposal, format specifiers appeared for priority and start time. The former was not defined adequately in this volume of IEEE Std 1003.1-200x and was removed in deference to the defined nice value; the latter because elapsed time was considered to be more useful.

In a new BSD version of *ps*, a −**O** option can be used to write all of the default information, followed by additional format specifiers. This was not adopted because the default output is implementation-defined. Nevertheless, this is a useful option that should be reserved for that purpose. In the −**o** option for the POSIX Shell and Utilities *ps*, the format is the concatenation of each −**o**. Therefore, the user can have an alias or function that defines the beginning of their desired format and add more fields to the end of the output in certain cases where that would be useful.

The format of the terminal name is unspecified, but the descriptions of *ps*, *talk*, *who*, and *write* require that they all use the same format.

The **pcpu** field indicates that the CPU time available is determined in an unspecified manner. This is because it is difficult to express an algorithm that is useful across all possible machine architectures. Historical counterparts to this value have attempted to show percentage of use in the recent past, such as the preceding minute. Frequently, these values for all processes did not add up to 100%. Implementations are encouraged to provide data in this field to users that will help them identify processes currently affecting the performance of the system.

FUTURE DIRECTIONS

None.

SEE ALSO

kill, *nice*, *renice*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

NAME

pwd — return working directory name

SYNOPSIS

```
pwd [−L | −P ]
```

DESCRIPTION

The *pwd* utility shall write to standard output an absolute pathname of the current working directory, which does not contain the filenames dot or dot-dot.

OPTIONS

The *pwd* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−L	If the *PWD* environment variable contains an absolute pathname of the current directory that does not contain the filenames dot or dot-dot, *pwd* shall write this pathname to standard output. Otherwise, the **−L** option shall behave as the **−P** option.
−P	The absolute pathname written shall not contain filenames that, in the context of the pathname, refer to files of type symbolic link.

If both **−L** and **−P** are specified, the last one shall apply. If neither **−L** nor **−P** is specified, the *pwd* utility shall behave as if **−L** had been specified.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *pwd*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PWD	If the **−P** option is in effect, this variable shall be set to an absolute pathname of the current working directory that does not contain any components that specify symbolic links, does not contain any components that are dot, and does not contain any components that are dot-dot. If an application sets or unsets the value of *PWD*, the behavior of *pwd* is unspecified.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *pwd* utility output is an absolute pathname of the current working directory:

`"%s\n", <directory pathname>`

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

If an error is detected, output shall not be written to standard output, a diagnostic message shall be written to standard error, and the exit status is not zero.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

Some implementations have historically provided *pwd* as a shell special built-in command.

In most utilities, if an error occurs, partial output may be written to standard output. This does not happen in historical implementations of *pwd*. Because *pwd* is frequently used in historical shell scripts without checking the exit status, it is important that the historical behavior is required here; therefore, the CONSEQUENCES OF ERRORS section specifically disallows any partial output being written to standard output.

FUTURE DIRECTIONS

None.

SEE ALSO

cd, the System Interfaces volume of IEEE Std 1003.1-200x, *getcwd*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

The –**P** and –**L** options are added to describe actions relating to symbolic links as specified in the IEEE P1003.2b draft standard.

NAME

qalter — alter batch job

SYNOPSIS

BE

```
qalter [−a date_time][−A account_string][−c interval][−e path_name]
    [−h hold_list][−j join_list][−k keep_list][−l resource_list]
    [−m mail_options][−M mail_list][−N name][−o path_name]
    [−p priority][−r y|n][−S path_name_list][−u user_list]
    job_identifier ...
```

DESCRIPTION

The attributes of a batch job are altered by a request to the batch server that manages the batch job. The *qalter* utility is a user-accessible batch client that requests the alteration of the attributes of one or more batch jobs.

The *qalter* utility shall alter the attributes of those batch jobs, and only those batch jobs, for which a batch *job_identifier* is presented to the utility.

The *qalter* utility shall alter the attributes of batch jobs in the order in which the batch *job_identifier*s are presented to the utility.

If the *qalter* utility fails to process a batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

For each batch *job_identifier* for which the *qalter* utility succeeds, each attribute of the identified batch job shall be altered as indicated by all the options presented to the utility.

For each identified batch job for which the *qalter* utility fails, the utility shall not alter any attribute of the batch job.

For each batch job that the *qalter* utility processes, the utility shall not modify any attribute other than those required by the options and option-arguments presented to the utility.

The *qalter* utility shall alter batch jobs by sending a *Modify Job Request* to the batch server that manages each batch job. At the time the *qalter* utility exits, it shall have modified the batch job corresponding to each successfully processed batch *job_identifier*. An attempt to alter the attributes of a batch job in the RUNNING state is implementation-defined.

OPTIONS

The *qalter* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−a *date_time* Redefine the time at which the batch job becomes eligible for execution.

The *date_time* argument shall be in the same form and represent the same time as for the *touch* utility. The time so represented shall be set into the *Execution_Time* attribute of the batch job. If the time specified is earlier than the current time, the **−a** option shall have no effect.

−A *account_string*

Redefine the account to which the resource consumption of the batch job should be charged.

The syntax of the *account_string* option-argument is unspecified.

The *qalter* utility shall set the *Account_Name* attribute of the batch job to the value of the *account_string* option-argument.

–**c** *interval* Redefine whether the batch job should be checkpointed, and if so, how often.

The *qalter* utility shall accept a value for the interval option-argument that is one of the following:

`n`	No checkpointing is to be performed on the batch batch job (NO_CHECKPOINT).
`s`	Checkpointing is to be performed only when the batch server is shut down (CHECKPOINT_AT_SHUTDOWN).
`c`	Automatic periodic checkpointing is to be performed at the *Minimum_Cpu_Interval* attribute of the batch queue, in units of CPU minutes (CHECKPOINT_AT_MIN_CPU_INTERVAL).
`c`=*minutes*	Automatic periodic checkpointing is to be performed every *minutes* of CPU time, or every *Minimum_Cpu_Interval* minutes, whichever is greater. The *minutes* argument shall conform to the syntax for unsigned integers and shall be greater than zero.

An implementation may define other checkpoint intervals. The conformance document for an implementation shall describe any alternative checkpoint intervals, how they are specified, their internal behavior, and how they affect the behavior of the utility.

The *qalter* utility shall set the *Checkpoint* attribute of the batch job to the value of the *interval* option-argument.

–**e** *path_name* Redefine the path to be used for the standard error stream of the batch job.

The *qalter* utility shall accept a *path_name* option-argument that conforms to the syntax of the *path_name* element defined in the System Interfaces volume of IEEE Std 1003.1-200x, which can be preceded by a host name element of the form *hostname*:.

If the *path_name* option-argument constitutes an absolute pathname, the *qalter* utility shall set the *Error_Path* attribute of the batch job to the value of the *path_name* option-argument, including the host name element, if present.

If the *path_name* option-argument constitutes a relative pathname and no host name element is specified, the *qalter* utility shall set the *Error_Path* attribute of the batch job to the value of the absolute pathname derived by expanding the *path_name* option-argument relative to the current directory of the process that executes the *qalter* utility.

If the *path_name* option-argument constitutes a relative pathname and a host name element is specified, the *qalter* utility shall set the *Error_Path* attribute of the batch job to the value of the option-argument without expansion.

If the *path_name* option-argument does not include a host name element, the *qalter* utility shall prefix the pathname in the *Error_Path* attribute with *hostname*:, where *hostname* is the name of the host upon which the *qalter* utility is being executed.

–**h** *hold_list* Redefine the types of holds, if any, on the batch job. The *qalter* –**h** option shall accept a value for the *hold_list* option-argument that is a string of alphanumeric characters in the portable character set.

The *qalter* utility shall accept a value for the *hold_list* option-argument that is a string of one or more of the characters `'u'`, `'s'`, or `'o'`, or the single character `'n'`. For each unique character in the *hold_list* option-argument, the *qalter* utility shall add a value to the *Hold_Types* attribute of the batch job as follows, each representing a different hold type:

`u` USER

`s` SYSTEM

`o` OPERATOR

If any of these characters are duplicated in the *hold_list* option-argument, the duplicates shall be ignored. An existing *Hold_Types* attribute can be cleared by the hold type:

`n` NO_HOLD

The *qalter* utility shall consider it an error if any hold type other than `'n'` is combined with hold type `'n'`. Strictly conforming applications shall not repeat any of the characters `'u'`, `'s'`, `'o'`, or `'n'` within the *hold_list* option-argument. The *qalter* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters. An implementation may define other hold types. The conformance document for an implementation shall describe any additional hold types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

−**j** *join_list* Redefine which streams of the batch job are to be merged. The *qalter* −**j** option shall accept a value for the *join_list* option-argument that is a string of alphanumeric characters in the portable character set.

The *qalter* utility shall accept a *join_list* option-argument that consists of one or more of the characters `'e'` and `'o'`, or the single character `'n'`.

All of the other batch job output streams specified shall be merged into the output stream represented by the character listed first in the *join_list* option-argument.

For each unique character in the *join_list* option-argument, the *qalter* utility shall add a value to the *Join_Path* attribute of the batch job as follows, each representing a different batch job stream to join:

`e` The standard error of the batch batch job (JOIN_STD_ERROR).

`o` The standard output of the batch batch job (JOIN_STD_OUTPUT).

An existing *Join_Path* attribute can be cleared by the join type:

`n` NO_JOIN

If `'n'` is specified, then no files are joined. The *qalter* utility shall consider it an error if any join type other than `'n'` is combined with join type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'e'`, `'o'`, or `'n'` within the *join_list* option-argument. The *qalter* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters.

An implementation may define other join types. The conformance document for an implementation shall describe any additional batch job streams, how they are specified, their internal behavior, and how they affect the behavior of the utility.

−**k** *keep_list* Redefine which output of the batch job to retain on the execution host.

The *qalter* −**k** option shall accept a value for the *keep_list* option-argument that is a string of alphanumeric characters in the portable character set.

The *qalter* utility shall accept a *keep_list* option-argument that consists of one or more of the characters `'e'` and `'o'` or the single character `'n'`.

For each unique character in the *keep_list* option-argument, the *qalter* utility shall add a value to the *Keep_Files* attribute of the batch job as follows, each representing a different batch job stream to keep:

`e` The standard error of the batch batch job (KEEP_STD_ERROR).

`o` The standard output of the batch batch job (KEEP_STD_OUTPUT).

If both `'e'` and `'o'` are specified, then both files are retained. An existing *Keep_Files* attribute can be cleared by the keep type:

`n` NO_KEEP

If `'n'` is specified, then no files are retained. The *qalter* utility shall consider it an error if any keep type other than `'n'` is combined with keep type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'e'`, `'o'`, or `'n'` within the *keep_list* option-argument. The *qalter* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters. An implementation may define other keep types. The conformance document for an implementation shall describe any additional keep types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

−l *resource_list*

Redefine the resources that are allowed or required by the batch job.

The *qalter* utility shall accept a *resource_list* option-argument that conforms to the following syntax:

```
resource=value[,,resource=value,,...]
```

The *qalter* utility shall set one entry in the value of the *Resource_List* attribute of the batch job for each resource listed in the *resource_list* option-argument.

Because the list of supported resource names might vary by batch server, the *qalter* utility shall rely on the batch server to validate the resource names and associated values. See Section 3.3.3 (on page 2325) for a means of removing *keyword=value* (and *value@keyword*) pairs and other general rules for list-oriented batch job attributes.

−m *mail_options*

Redefine the points in the execution of the batch job at which the batch server is to send mail about a change in the state of the batch job.

The *qalter* **−m** option shall accept a value for the *mail_options* option-argument that is a string of alphanumeric characters in the portable character set.

The *qalter* utility shall accept a value for the *mail_options* option-argument that is a string of one or more of the characters `'e'`, `'b'`, and `'a'`, or the single character `'n'`. For each unique character in the *mail_options* option-argument, the *qalter* utility shall add a value to the *Mail_Users* attribute of the batch job as follows, each representing a different time during the life of a batch job at which to send mail:

`e` MAIL_AT_EXIT

`b` MAIL_AT_BEGINNING

`a` MAIL_AT_ABORT

If any of these characters are duplicated in the *mail_options* option-argument, the duplicates shall be ignored.

An existing *Mail_Points* attribute can be cleared by the mail type:

`n` NO_MAIL

If `'n'` is specified, then mail is not sent. The *qalter* utility shall consider it an error if any mail type other than `'n'` is combined with mail type `'n'`. Strictly conforming applications shall not repeat any of the characters `'e'`, `'b'`, `'a'`, or `'n'` within the *mail_options* option-argument. The *qalter* utility shall permit the repetition of characters but shall not assign additional meaning to the repeated characters.

An implementation may define other mail types. The conformance document for an implementation shall describe any additional mail types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

−**M** *mail_list* Redefine the list of users to which the batch server that executes the batch job is to send mail, if the batch server sends mail about the batch job.

The syntax of the *mail_list* option-argument is unspecified. If the implementation of the *qalter* utility uses a name service to locate users, the utility shall accept the syntax used by the name service.

If the implementation of the *qalter* utility does not use a name service to locate users, the implementation shall accept the following syntax for user names:

```
mail_address[,,mail_address,,...]
```

The interpretation of *mail_address* is implementation-defined.

The *qalter* utility shall set the *Mail_Users* attribute of the batch job to the value of the *mail_list* option-argument.

−**N** *name* Redefine the name of the batch job.

The *qalter* −**N** option shall accept a value for the *name* option-argument that is a string of up to 15 alphanumeric characters in the portable character set where the first character is alphabetic.

The syntax of the *name* option-argument is unspecified.

The *qalter* utility shall set the *Job_Name* attribute of the batch job to the value of the *name* option-argument.

−**o** *path_name* Redefine the path for the standard output of the batch job.

The *qalter* utility shall accept a *path_name* option-argument that conforms to the syntax of the *path_name* element defined in the System Interfaces volume of IEEE Std 1003.1-200x, which can be preceded by a host name element of the form *hostname*:.

If the *path_name* option-argument constitutes an absolute pathname, the *qalter* utility shall set the *Output_Path* attribute of the batch job to the value of the *path_name* option-argument.

If the *path_name* option-argument constitutes a relative pathname and no host name element is specified, the *qalter* utility shall set the *Output_Path* attribute of the batch job to the absolute pathname derived by expanding the *path_name* option-argument relative to the current directory of the process that executes the *qalter* utility.

If the *path_name* option-argument constitutes a relative pathname and a host name element is specified, the *qalter* utility shall set the *Output_Path* attribute of the batch job to option-argument without any expansion of the pathname.

If the *path_name* option-argument does not include a host name element, the *qalter* utility shall prefix the pathname in the *Output_Path* attribute with *hostname*:, where *hostname* is the name of the host upon which the *qalter* utility is being executed.

−**p** *priority* Redefine the priority of the batch job.

The *qalter* utility shall accept a value for the priority option-argument that conforms to the syntax for signed decimal integers, and which is not less than −1 024 and not greater than 1 023.

The *qalter* utility shall set the *Priority* attribute of the batch job to the value of the *priority* option-argument.

−**r** `y|n` Redefine whether the batch job is rerunable.

If the value of the option-argument is `'y'`, the *qalter* utility shall set the *Rerunable* attribute of the batch job to TRUE.

If the value of the option-argument is `'n'`, the *qalter* utility shall set the *Rerunable* attribute of the batch job to FALSE.

The *qalter* utility shall consider it an error if any character other than `'y'` or `'n'` is specified in the option-argument.

−**S** *path_name_list*

Redefine the shell that interprets the script at the destination system.

The *qalter* utility shall accept a *path_name_list* option-argument that conforms to the following syntax:

```
pathname[@host][,pathname[@host],...]
```

The *qalter* utility shall accept only one pathname that is missing a corresponding host name. The *qalter* utility shall allow only one pathname per named host.

The *qalter* utility shall add a value to the *Shell_Path_List* attribute of the batch job for each entry in the *path_name_list* option-argument. See Section 3.3.3 (on page 2325) for a means of removing *keyword*=*value* (and *value*@*keyword*) pairs and other general rules for list-oriented batch job attributes.

−**u** *user_list* Redefine the user name under which the batch job is to run at the destination system.

The *qalter* utility shall accept a *user_list* option-argument that conforms to the following syntax:

```
username[@host][,,username[@host],,...]
```

The *qalter* utility shall accept only one user name that is missing a corresponding host name. The *qalter* utility shall accept only one user name per named host.

The *qalter* utility shall add a value to the *User_List* attribute of the batch job for each entry in the *user_list* option-argument. See Section 3.3.3 (on page 2325) for a means of removing *keyword*=*value* (and *value*@*keyword*) pairs and other general rules for list-oriented batch job attributes.

OPERANDS

The *qalter* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qalter*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.
TZ	Determine the timezone used to interpret the *date-time* option-argument. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qalter* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qalter* utility attempts to locate the batch job on other batch servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qalter* utility allows users to change the attributes of a batch job.

As a means of altering a queued job, the *qalter* utility is superior to deleting and requeuing the batch job insofar as an altered job retains its place in the queue with some traditional selection algorithms. In addition, the *qalter* utility is both shorter and simpler than a sequence of *qdel* and *qsub* utilities.

The result of an attempt on the part of a user to alter a batch job in a RUNNING state is implementation-defined because a batch job in the RUNNING state will already have opened its output files and otherwise performed any actions indicated by the options in effect at the time the batch job began execution.

The options processed by the *qalter* utility are identical to those of the *qsub* utility, with a few exceptions: −**V**, −**v**, and −**q**. The −**V** and −**v** are inappropriate for the *qalter* utility, since they capture potentially transient environment information from the submitting process. The −**q** option would specify a new queue, which would largely negate the previously stated advantage of using *qalter*; furthermore, the *qmove* utility provides a superior means of moving jobs.

Each of the following paragraphs provides the rationale for a *qalter* option.

Additional rationale concerning these options can be found in the rationale for the *qsub* utility.

The −**a** option allows users to alter the date and time at which a batch job becomes eligible to run.

The −**A** option allows users to change the account that will be charged for the resources consumed by the batch job. Support for the −**A** option is mandatory for conforming implementations of *qalter*, even though support of accounting is optional for servers. Whether or not to support accounting is left to the implementor of the server, but mandatory support of the −**A** option assures users of a consistent interface and allows them to control accounting on servers that support accounting.

The −**c** option allows users to alter the checkpointing interval of a batch job. A checkpointing system, which is not defined by IEEE Std 1003.1-200x, allows recovery of a batch job at the most recent checkpoint in the event of a crash. Checkpointing is typically used for jobs that consume expensive computing time or must meet a critical schedule. Users should be allowed to make the tradeoff between the overhead of checkpointing and the risk to the timely completion of the batch job; therefore, this volume of IEEE Std 1003.1-200x provides the checkpointing interval option. Support for checkpointing is optional for servers.

The −**e** option allows users to alter the name and location of the standard error stream written by a batch job. However, the path of the standard error stream is meaningless if the value of the *Join_Path* attribute of the batch job is TRUE.

The −**h** option allows users to set the hold type in the *Hold_Types* attribute of a batch job. The *qhold* and *qrls* utilities add or remove hold types to the *Hold_Types* attribute, respectively. The −**h** option has been modified to allow for implementation-defined hold types.

The −**j** option allows users to alter the decision to join (merge) the standard error stream of the batch job with the standard output stream of the batch job.

The –**l** option allows users to change the resource limits imposed on a batch job.

The –**m** option allows users to modify the list of points in the life of a batch job at which the designated users will receive mail notification.

The –**M** option allows users to alter the list of users who will receive notification about events in the life of a batch job.

The –**N** option allows users to change the name of a batch job.

The –**o** option allows users to alter the name and path to which the standard output stream of the batch job will be written.

The –**P** option allows users to modify the priority of a batch job. Support for priority is optional for batch servers.

The –**r** option allows users to alter the rerunability status of a batch job.

The –**S** option allows users to change the name and location of the shell image that will be invoked to interpret the script of the batch job. This option has been modified to allow a list of shell name and locations associated with different host.

The –**u** option allows users to change the user identifier under which the batch job will execute.

The *job_identifier* operand syntax is provided so that the user can differentiate between the originating and destination (or executing) batch server. These may or may not be the same. The *.server_name* portion identifies the originating batch server, while the *@server* portion identifies the destination batch server.

Historically, the *qalter* utility has been a component of the Network Queuing System (NQS), the existing practice from which this utility has been derived.

FUTURE DIRECTIONS

None.

SEE ALSO

qdel, *qhold*, *qmove*, *qrls*, *qsub*, *touch*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

IEEE PASC Interpretation 1003.2 #182 is applied, clarifying the description of the –**a** option.

NAME

qdel — delete batch jobs

SYNOPSIS

BE
```
qdel job_identifier ...
```

DESCRIPTION

A batch job is deleted by sending a request to the batch server that manages the batch job. A batch job that has been deleted is no longer subject to management by batch services.

The *qdel* utility is a user-accessible client of batch services that requests the deletion of one or more batch jobs.

The *qdel* utility shall request a batch server to delete those batch jobs for which a batch *job_identifier* is presented to the utility.

The *qdel* utility shall delete batch jobs in the order in which their batch *job_identifier*s are presented to the utility.

If the *qdel* utility fails to process any batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qdel* utility shall delete each batch job by sending a *Delete Job Request* to the batch server that manages the batch job.

The *qdel* utility shall not exit until the batch job corresponding to each successfully processed batch *job_identifier* has been deleted.

OPTIONS

None.

OPERANDS

The *qdel* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qdel*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LOGNAME Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

An implementation of the *qdel* utility may write informative messages to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qdel* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qdel* utility waits to output the diagnostic message while attempting to locate the job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qdel* utility allows users and administrators to delete jobs.

The *qdel* utility provides functionality that is not otherwise available. For example, the *kill* utility of the operating system does not suffice. First, to use the *kill* utility, the user might have to log in on a remote node, because the *kill* utility does not operate across the network. Second, unlike *qdel*, *kill* cannot remove jobs from queues. Lastly, the arguments of the *qdel* utility are job identifiers rather than process identifiers, and so this utility can be passed the output of the *qselect* utility, thus providing users with a means of deleting a list of jobs.

Because a set of jobs can be selected using the *qselect* utility, the *qdel* utility has not been complicated with options that provide for selection of jobs. Instead, the batch jobs to be deleted are identified individually by their job identifiers.

Historically, the *qdel* utility has been a component of NQS, the existing practice on which it is based. However, the *qdel* utility defined in this volume of IEEE Std 1003.1-200x does not provide an option for specifying a signal number to send to the batch job prior to the killing of the process; that capability has been subsumed by the *qsig* utility.

A discussion was held about the delays of networking and the possibility that the batch server may never respond, due to a down router, down batch server, or other network mishap. The DESCRIPTION records this under the words ‘‘fails to process any job identifier’’. In the broad sense, the network problem is also an error, which causes the failure to process the batch job identifier.

FUTURE DIRECTIONS

None.

SEE ALSO

kill, *qselect*, *qsig*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qhold — hold batch jobs

SYNOPSIS

BE
```
qhold [−h hold_list] job_identifier ...
```

DESCRIPTION

A hold is placed on a batch job by a request to the batch server that manages the batch job. A batch job that has one or more holds is not eligible for execution. The *qhold* utility is a user-accessible client of batch services that requests one or more types of hold to be placed on one or more batch jobs.

The *qhold* utility shall place holds on those batch jobs for which a batch *job_identifier* is presented to the utility.

The *qhold* utility shall place holds on batch jobs in the order in which their batch *job_identifier*s are presented to the utility. If the *qhold* utility fails to process any batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qhold* utility shall place holds on each batch job by sending a *Hold Job Request* to the batch server that manages the batch job.

The *qhold* utility shall not exit until holds have been placed on the batch job corresponding to each successfully processed batch *job_identifier*.

OPTIONS

The *qhold* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported by the implementation:

−**h** *hold_list* Define the types of holds to be placed on the batch job.

The *qhold* −**h** option shall accept a value for the *hold_list* option-argument that is a string of alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

The *qhold* utility shall accept a value for the *hold_list* option-argument that is a string of one or more of the characters `'u'`, `'s'`, or `'o'`, or the single character `'n'`.

For each unique character in the *hold_list* option-argument, the *qhold* utility shall add a value to the *Hold_Types* attribute of the batch job as follows, each representing a different hold type:

`u`	USER
`s`	SYSTEM
`o`	OPERATOR

If any of these characters are duplicated in the *hold_list* option-argument, the duplicates shall be ignored.

An existing *Hold_Types* attribute can be cleared by the following hold type:

`n` NO_HOLD

The *qhold* utility shall consider it an error if any hold type other than `'n'` is combined with hold type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'u'`, `'s'`, `'o'`, or `'n'` within the *hold_list* option-argument. The *qhold* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters.

An implementation may define other hold types. The conformance document for an implementation shall describe any additional hold types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

If the −**h** option is not presented to the *qhold* utility, the implementation shall set the *Hold_Types* attribute to USER.

OPERANDS

The *qhold* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qhold*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qhold* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qhold* utility waits to output the diagnostic message while attempting to locate the job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qhold* utility allows users to place a hold on one or more jobs. A hold makes a batch job ineligible for execution.

The *qhold* utility has options that allow the user to specify the type of hold. Should the user wish to place a hold on a set of jobs that meet a selection criteria, such a list of jobs can be acquired using the *qselect* utility.

The −**h** option allows the user to specify the type of hold that is to be placed on the job. This option allows for USER, SYSTEM, OPERATOR, and implementation-defined hold types. The USER and OPERATOR holds are distinct. The batch server that manages the batch job will verify that the user is authorized to set the specified hold for the batch job.

Mail is not required on hold because the administrator has the tools and libraries to build this option if he or she wishes.

Historically, the *qhold* utility has been a part of some existing batch systems, although it has not traditionally been a part of the NQS.

FUTURE DIRECTIONS

None.

SEE ALSO

qselect, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qmove — move batch jobs

SYNOPSIS

BE `qmove destination job_identifier ...`

DESCRIPTION

To move a batch job is to remove the batch job from the batch queue in which it resides and instantiate the batch job in another batch queue. A batch job is moved by a request to the batch server that manages the batch job. The *qmove* utility is a user-accessible batch client that requests the movement of one or more batch jobs.

The *qmove* utility shall move those batch jobs, and only those batch jobs, for which a batch *job_identifier* is presented to the utility.

The *qmove* utility shall move batch jobs in the order in which the corresponding batch *job_identifier*s are presented to the utility.

If the *qmove* utility fails to process a batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qmove* utility shall move batch jobs by sending a *Move Job Request* to the batch server that manages each batch job. The *qmove* utility shall not exit before the batch jobs corresponding to all successfully processed batch *job_identifier*s have been moved.

OPTIONS

None.

OPERANDS

The *qmove* utility shall accept one operand that conforms to the syntax for a *destination* (see Section 3.3.2 (on page 2325)).

The *qmove* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qmove*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LOGNAME	Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qmove* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qmove* utility waits to output the diagnostic message while attempting to locate the job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qmove* utility allows users to move jobs between queues.

The alternative to using the *qmove* utility—deleting the batch job and requeuing it—entails considerably more typing.

Since the means of selecting jobs based on attributes has been encapsulated in the *qselect* utility, the only option of the *qmove* utility concerns authorization. The −**u** option provides the user with the convenience of changing the user identifier under which the batch job will execute. Minimalism and consistency has taken precedence over convenience; the −**u** option has been deleted because the equivalent capability exists with the −**u** option of the *qalter* utility.

FUTURE DIRECTIONS

None.

SEE ALSO

qalter, *qselect*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qmsg — send message to batch jobs

SYNOPSIS

BE `qmsg [−E][−O] message_string job_identifier ...`

DESCRIPTION

To send a message to a batch job is to request that a server write a message string into one or more output files of the batch job. A message is sent to a batch job by a request to the batch server that manages the batch job. The *qmsg* utility is a user-accessible batch client that requests the sending of messages to one or more batch jobs.

The *qmsg* utility shall write messages into the files of batch jobs by sending a *Job Message Request* to the batch server that manages the batch job. The *qmsg* utility shall not directly write the message into the files of the batch job.

The *qmsg* utility shall send a *Job Message Request* for those batch jobs, and only those batch jobs, for which a batch *job_identifier* is presented to the utility.

The *qmsg* utility shall send *Job Message Request*s for batch jobs in the order in which their batch *job_identifier*s are presented to the utility.

If the *qmsg* utility fails to process any batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qmsg* utility shall not exit before a *Job Message Request* has been sent to the server that manages the batch job that corresponds to each successfully processed batch *job_identifier*.

OPTIONS

The *qmsg* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−E Specify that the message is written to the standard error of each batch job.

The *qmsg* utility shall write the message into the standard error of the batch job.

−O Specify that the message is written to the standard output of each batch job.

The *qmsg* utility shall write the message into the standard output of the batch job.

If neither the −**O** nor the −**E** option is presented to the *qmsg* utility, the utility shall write the message into an implementation-defined file. The conformance document for the implementation shall describe the name and location of the implementation-defined file. If both the −**O** and the −**E** options are presented to the *qmsg* utility, then the utility shall write the messages to both standard output and standard error.

OPERANDS

The *qmsg* utility shall accept a minimum of two operands, *message_string* and one or more batch *job_identifier*s.

The *message_string* operand shall be the string to be written to one or more output files of the batch job followed by a <newline>. If the string contains <blank>s, then the application shall ensure that the string is quoted. The *message_string* shall be encoded in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

All remaining operands are batch *job_identifier*s that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qmsg*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qmsg* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qmsg* utility waits to output the diagnostic message while attempting to locate the job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qmsg* utility allows users to write messages into the output files of running jobs. Users, including operators and administrators, have a number of occasions when they want to place messages in the output files of a batch job. For example, if a disk that is being used by a batch job is showing errors, the operator might note this in the standard error stream of the batch job.

The options of the *qmsg* utility provide users with the means of placing the message in the output stream of their choice. The default output stream for the message—if the user does not designate an output stream—is implementation-defined, since many implementations will provide, as an extension to this volume of IEEE Std 1003.1-200x, a log file that shows the history of utility execution.

If users wish to send a message to a set of jobs that meet a selection criteria, the *qselect* utility can be used to acquire the appropriate list of job identifiers.

The −**E** option allows users to place the message in the standard error stream of the batch job.

The −**O** option allows users to place the message in the standard output stream of the batch job.

Historically, the *qmsg* utility is an existing practice in the offerings of one or more implementors of an NQS-derived batch system. The utility has been found to be useful enough that it deserves to be included in this volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

qselect, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qrerun — rerun batch jobs

SYNOPSIS

BE

```
qrerun job_identifier ...
```

DESCRIPTION

To rerun a batch job is to terminate the session leader of the batch job, delete any associated checkpoint files, and return the batch job to the batch queued state. A batch job is rerun by a request to the batch server that manages the batch job. The *qrerun* utility is a user-accessible batch client that requests the rerunning of one or more batch jobs.

The *qrerun* utility shall rerun those batch jobs for which a batch *job_identifier* is presented to the utility.

The *qrerun* utility shall rerun batch jobs in the order in which their batch *job_identifier*s are presented to the utility.

If the *qrerun* utility fails to process any batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qrerun* utility shall rerun batch jobs by sending a *Rerun Job Request* to the batch server that manages each batch job.

For each successfully processed batch *job_identifier*, the *qrerun* utility shall have rerun the corresponding batch batch job at the time the utility exits.

OPTIONS

None.

OPERANDS

The *qrerun* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qrerun*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LOGNAME Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qrerun* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qrerun* utility waits to output the diagnostic message while attempting to locate the job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qrerun* utility allows users to cause jobs in the running state to exit and rerun.

The *qrerun* utility is a new utility, *vis-a-vis* existing practice, that has been defined in this volume of IEEE Std 1003.1-200x to correct user-perceived deficiencies in the existing practice.

FUTURE DIRECTIONS

None.

SEE ALSO

Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qrls — release batch jobs

SYNOPSIS

BE

```
qrls [−h hold_list] job_identifier ...
```

DESCRIPTION

A batch job might have one or more holds, which prevent the batch job from executing. A batch job from which all the holds have been removed becomes eligible for execution and is said to have been released. A batch job hold is removed by sending a request to the batch server that manages the batch job. The *qrls* utility is a user-accessible client of batch services that requests holds be removed from one or more batch jobs.

The *qrls* utility shall remove one or more holds from those batch jobs for which a batch *job_identifier* is presented to the utility.

The *qrls* utility shall remove holds from batch jobs in the order in which their batch *job_identifier*s are presented to the utility.

If the *qrls* utility fails to process a batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qrls* utility shall remove holds on each batch job by sending a *Release Job Request* to the batch server that manages the batch job.

The *qrls* utility shall not exit until the holds have been removed from the batch job corresponding to each successfully processed batch *job_identifier*.

OPTIONS

The *qrls* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported by the implementation:

−**h** *hold_list* Define the types of holds to be removed from the batch job.

The *qrls* −**h** option shall accept a value for the *hold_list* option-argument that is a string of alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

The *qrls* utility shall accept a value for the *hold_list* option-argument that is a string of one or more of the characters `'u'`, `'s'`, or `'o'`, or the single character `'n'`.

For each unique character in the *hold_list* option-argument, the *qrls* utility shall add a value to the *Hold_Types* attribute of the batch job as follows, each representing a different hold type:

`u`	USER
`s`	SYSTEM
`o`	OPERATOR

If any of these characters are duplicated in the *hold_list* option-argument, the duplicates shall be ignored.

An existing *Hold_Types* attribute can be cleared by the following hold type:

`n`	NO_HOLD

The *qrls* utility shall consider it an error if any hold type other than `'n'` is combined with hold type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'u'`, `'s'`, `'o'`, or `'n'` within the *hold_list* option-argument. The *qrls* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters.

An implementation may define other hold types. The conformance document for an implementation shall describe any additional hold types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

If the **−h** option is not presented to the *qrls* utility, the implementation shall remove the USER hold in the *Hold_Types* attribute.

OPERANDS

The *qrls* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qrls*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qrls* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qrls* utility waits to output the diagnostic message while attempting to locate the job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qrls* utility allows users, operators, and administrators to remove holds from jobs.

The *qrls* utility does not support any job selection options or wildcard arguments. Users may acquire a list of jobs selected by attributes using the *qselect* utility. For example, a user could select all of their held jobs.

The −**h** option allows the user to specify the type of hold that is to be removed. This option allows for USER, SYSTEM, OPERATOR, and implementation-defined hold types. The batch server that manages the batch job will verify whether the user is authorized to remove the specified hold for the batch job. If more than one type of hold has been placed on the batch job, a user may wish to remove only some of them.

Mail is not required on release because the administrator has the tools and libraries to build this option if required.

The *qrls* utility is a new utility *vis-a-vis* existing practice; it has been defined in this volume of IEEE Std 1003.1-200x as the natural complement to the *qhold* utility.

FUTURE DIRECTIONS

None.

SEE ALSO

qhold, *qselect*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qselect — select batch jobs

SYNOPSIS

BE

```
qselect [−a [op]date_time][−A account_string][−c [op]interval]
    [−h hold_list][−l resource_list][−N name][−p [op]priority]
    [−q destination][−r y|n][−s states][−u user_list]
```

DESCRIPTION

To select a set of batch jobs is to return the batch *job_identifier*s for each batch job that meets a list of selection criteria. A set of batch jobs is selected by a request to a batch server. The *qselect* utility is a user-accessible batch client that requests the selection of batch jobs.

Upon successful completion, the *qselect* utility shall have returned a list of zero or more batch *job_identifier*s that meet the criteria specified by the options and option-arguments presented to the utility.

The *qselect* utility shall select batch jobs by sending a *Select Jobs Request* to a batch server. The *qselect* utility shall not exit until the server replies to each request generated.

For each option presented to the *qselect* utility, the utility shall restrict the set of selected batch jobs as described in the OPTIONS section.

The *qselect* utility shall not restrict selection of batch jobs except by authorization and as required by the options presented to the utility.

When an option is specified with a mandatory or optional *op* component to the option-argument, then *op* shall specify a relation between the value of a certain batch job attribute and the *value* component of the option-argument. If an *op* is allowable on an option, then the description of the option letter indicates the *op* as either mandatory or optional. Acceptable strings for the *op* component, and the relation the string indicates, are shown in the following list:

`.eq.`	The value represented by the attribute of the batch job is equal to the value represented by the option-argument.
`.ge.`	The value represented by the attribute of the batch job is greater than or equal to the value represented by the option-argument.
`.gt.`	The value represented by the attribute of the batch job is greater than the value represented by the option-argument.
`.lt.`	The value represented by the attribute of the batch job is less than the value represented by the option-argument.
`.le.`	The value represented by the attribute of the batch job is less than or equal to the value represented by the option-argument.
`.ne.`	The value represented by the attribute of the batch job is not equal to the value represented by the option-argument.

OPTIONS

The *qselect* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−a [*op***]***date_time*

Restrict selection to a specific time, or a range of times.

The *qselect* utility shall select only batch jobs for which the value of the *Execution_Time* attribute is related to the Epoch equivalent of the local time expressed by the value of the *date_time* component of the option-argument in the manner indicated by the value of the *op* component of the option-argument.

The *qselect* utility shall accept a *date_time* component of the option-argument that conforms to the syntax of the *time* operand of the *touch* utility.

If the *op* component of the option-argument is not presented to the *qselect* utility, the utility shall select batch jobs for which the *Execution_Time* attribute is equal to the *date_time* component of the option-argument.

When comparing times, the *qselect* utility shall use the following definitions for the *op* component of the option-argument:

`.eq.` The time represented by value of the *Execution_Time* attribute of the batch job is equal the time represented by the *date_time* component of the option-argument.

`.ge.` The time represented by value of the *Execution_Time* attribute of the batch job is after or equal to the time represented by the *date_time* component of the option-argument.

`.gt.` The time represented by value of the *Execution_Time* attribute of the batch job is after the time represented by the *date_time* component of the option-argument.

`.lt.` The time represented by value of the *Execution_Time* attribute of the batch job is before the time represented by the *date_time* component of the option-argument.

`.le.` The time represented by value of the *Execution_Time* attribute of the batch job is before or equal to the time represented by the *date_time* component of the option-argument.

`.ne.` The time represented by value of the *Execution_Time* attribute of the batch job is not equal to the time represented by the *date_time* component of the option-argument.

The *qselect* utility shall accept the defined character strings for the *op* component of the option-argument.

−**A** *account_string*

Restrict selection to the batch jobs charging a specified account.

The *qselect* utility shall select only batch jobs for which the value of the *Account_Name* attribute of the batch job matchs the value of the *account_string* option-argument.

The syntax of the *account_string* option-argument is unspecified.

−**c** **[***op***]***interval*

Restrict selection to batch jobs within a range of checkpoint intervals.

The *qselect* utility shall select only batch jobs for which the value of the *Checkpoint* attribute relates to the value of the *interval* component of the option-argument in the manner indicated by the value of the *op* component of the option-argument.

If the *op* component of the option-argument is omitted, the *qselect* utility shall select batch jobs for which the value of the *Checkpoint* attribute is equal to the value of the *interval* component of the option-argument.

When comparing checkpoint intervals, the *qselect* utility shall use the following definitions for the *op* component of the option-argument:

`.eq.` The value of the *Checkpoint* attribute of the batch job equals the value of the *interval* component of the option-argument.

`.ge.` The value of the *Checkpoint* attribute of the batch job is greater than or equal to the value of the *interval* component option-argument.

`.gt.` The value of the *Checkpoint* attribute of the batch job is greater than the value of the *interval* component option-argument.

`.lt.` The value of the *Checkpoint* attribute of the batch job is less than the value of the *interval* component option-argument.

`.le.` The value of the *Checkpoint* attribute of the batch job is less than or equal to the value of the *interval* component option-argument.

`.ne.` The value of the *Checkpoint* attribute of the batch job does not equal the value of the *interval* component option-argument.

The *qselect* utility shall accept the defined character strings for the *op* component of the option-argument.

The ordering relationship for the values of the interval option-argument is defined to be:

`'n' .gt. 's' .gt. 'c=`*`minutes`*`' .ge. 'c'`

When comparing *Checkpoint* attributes with an interval having the value of the single character `'u'`, only equality or inequality are valid comparisons.

–h *hold_list* Restrict selection to batch jobs that have a specific type of hold.

The *qselect* utility shall select only batch jobs for which the value of the *Hold_Types* attribute matches the value of the *hold_list* option-argument.

The *qselect* **–h** option shall accept a value for the *hold_list* option-argument that is a string of alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

The *qselect* utility shall accept a value for the *hold_list* option-argument that is a string of one or more of the characters `'u'`, `'s'`, or `'o'`, or the single character `'n'`.

Each unique character in the *hold_list* option-argument of the *qselect* utility is defined as follows, each representing a different hold type:

`u` USER

`s` SYSTEM

`o` OPERATOR

If any of these characters are duplicated in the *hold_list* option-argument, the duplicates shall be ignored.

The *qselect* utility shall consider it an error if any hold type other than `'n'` is combined with hold type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'u'`, `'s'`, `'o'`, or `'n'` within the *hold_list* option-argument. The *qselect* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters.

An implementation may define other hold types. The conformance document for an implementation shall describe any additional hold types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

−l *resource_list*

Restrict selection to batch jobs with specified resource limits and attributes.

The *qselect* utility shall accept a *resource_list* option-argument with the following syntax:

`resource_name op value [,,resource_name op value,, ...]`

When comparing resource values, the *qselect* utility shall use the following definitions for the *op* component of the option-argument:

`.eq.`	The value of the resource of the same name in the *Resource_List* attribute of the batch job equals the value of the value component of the option-argument.
`.ge.`	The value of the resource of the same name in the *Resource_List* attribute of the batch job is greater than or equal to the value of the *value* component of the option-argument.
`.gt.`	The value of the resource of the same name in the *Resource_List* attribute of the batch job is greater than the value of the value component of the option-argument.
`.lt.`	The value of the resource of the same name in the *Resource_List* attribute of the batch job is less than the value of the value component of the option-argument.
`.ne.`	The value of the resource of the same name in the *Resource_List* attribute of the batch job does not equal the value of the value component of the option-argument.
`.le.`	The value of the resource of the same name in the *Resource_List* attribute of the batch job is less than or equal to the value of the *value* component of the option-argument.

When comparing the limit of a *Resource_List* attribute with the *value* component of the option-argument, if the limit, the value, or both are non-numeric, only equality or inequality are valid comparisons.

The *qselect* utility shall select only batch jobs for which the values of the *resource_names* listed in the *resource_list* option-argument match the corresponding limits of the *Resource_List* attribute of the batch job.

Limits of *resource_names* present in the *Resource_List* attribute of the batch job that have no corresponding values in the *resource_list* option-argument shall not be considered when selecting batch jobs.

−N *name* Restrict selection to batch jobs with a specified name.

The *qselect* utility shall select only batch jobs for which the value of the *Job_Name* attribute matches the value of the *name* option-argument. The string specified in the *name* option-argument shall be passed, uninterpreted, to the server. This allows an implementation to match ‘‘wildcard’’ patterns against batch job names.

An implementation shall describe in the conformance document the format it supports for matching against the *Job_Name* attribute.

−p [*op***]***priority*

Restrict selection to batch jobs of the specified priority or range of priorities.

The *qselect* utility shall select only batch jobs for which the value of the *Priority* attribute of the batch job relates to the value of the *priority* component of the option-argument in the manner indicated by the value of the *op* component of the option-argument.

If the *op* component of the option-argument is omitted, the *qselect* utility shall select batch jobs for which the value of the *Priority* attribute of the batch job is equal to the value of the *priority* component of the option-argument.

When comparing priority values, the *qselect* utility shall use the following definitions for the *op* component of the option-argument:

`.eq.` The value of the *Priority* attribute of the batch job equals the value of the *priority* component of the option-argument.

`.ge.` The value of the *Priority* attribute of the batch job is greater than or equal to the value of the *priority* component option-argument.

`.gt.` The value of the *Priority* attribute of the batch job is greater than the value of the *priority* component option-argument.

`.lt.` The value of the *Priority* attribute of the batch job is less than the value of the *priority* component option-argument.

`.lt.` The value of the *Priority* attribute of the batch job is less than or equal to the value of the *priority* component option-argument.

`.ne.` The value of the *Priority* attribute of the batch job does not equal the value of the *priority* component option-argument.

−q *destination*

Restrict selection to the specified batch queue or server, or both.

The *qselect* utility shall select only batch jobs that are located at the destination indicated by the value of the *destination* option-argument.

The destination defines a batch queue, a server, or a batch queue at a server.

The *qselect* utility shall accept an option-argument for the **−q** option that conforms to the syntax for a destination. If the **−q** option is not presented to the *qselect* utility, the utility shall select batch jobs from all batch queues at the default batch server.

If the option-argument describes only a batch queue, the *qselect* utility shall select only batch jobs from the batch queue of the specified name at the default batch server. The means by which *qselect* determines the default server is implementation-defined.

If the option-argument describes only a batch server, the *qselect* utility shall select batch jobs from all the batch queues at that batch server.

If the option-argument describes both a batch queue and a batch server, the *qselect* utility shall select only batch jobs from the specified batch queue at the specified server.

−r `y|n` Restrict selection to batch jobs with the specified rerunability status.

The *qselect* utility shall select only batch jobs for which the value of the *Rerunable* attribute of the batch job matches the value of the option-argument.

The *qselect* utility shall accept a value for the option-argument that consists of either the single character `'y'` or the single character `'n'`. The character `'y'` represents the value TRUE, and the character `'n'` represents the value FALSE.

−s *states* Restrict selection to batch jobs in the specified states.

The *qselect* utility shall accept an option-argument that consists of any combination of the characters `'e'`, `'q'`, `'r'`, `'w'`, `'h'`, and `'t'`.

Conforming applications shall not repeat any character in the option-argument. The *qselect* utility shall permit the repetition of characters in the option-argument, but shall not assign additional meaning to repeated characters.

The *qselect* utility shall interpret the characters in the *states* option-argument as follows:

`e`	Represents the EXITING state.
`q`	Represents the QUEUED state.
`r`	Represents the RUNNING state.
`t`	Represents the TRANSITING state.
`h`	Represents the HELD state.
`w`	Represents the WAITING state.

For each character in the *states* option-argument, the *qselect* utility shall select batch jobs in the corresponding state.

−u *user_list* Restrict selection to batch jobs owned by the specified user names.

The *qselect* utility shall select only the batch jobs of those users specified in the *user_list* option-argument.

The *qselect* utility shall accept a *user_list* option-argument that conforms to the following syntax:

```
username[@host][,,username[@host],, ...]
```

The *qselect* utility shall accept only one user name that is missing a corresponding host name. The *qselect* utility shall accept only one user name per named host.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qselect*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.
TZ	Determine the timezone used to interpret the *date-time* option-argument. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *qselect* utility shall write zero or more batch *job_identifier*s to standard output.

The *qselect* utility shall separate the batch *job_identifier*s written to standard output by white space.

The *qselect* utility shall write batch *job_identifier*s in the following format:

```
sequence_number.server_name@server
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

The following example shows how a user might use the *qselect* utility in conjunction with the *qdel* utility to delete all of his or her jobs in the queued state without affecting any jobs that are already running:

```
qdel $(qselect −s q)
```

or:

```
qselect −s q || xargs qdel
```

RATIONALE

The *qselect* utility allows users to acquire a list of job identifiers that match user-specified selection criteria. The list of identifiers returned by the *qselect* utility conforms to the syntax of the batch job identifier list processed by a utility such as *qmove*, *qdel*, and *qrls*.The *qselect* utility is thus a powerful tool for causing another batch system utility to act upon a set of jobs that match a list of selection criteria.

The options of the *qselect* utility let the user apply a number of useful filters for selecting jobs. Each option further restricts the selection of jobs. Many of the selection options allow the specification of a relational operator. The FORTRAN-like syntax of the operator—that is, `".lt."`, was chosen rather than the C-like "<=" meta-characters.

The −**a** option allows users to restrict the selected jobs to those that have been submitted (or altered) to wait until a particular time. The time period is determined by the argument of this option, which includes both a time and an operator—it is thus possible to select jobs waiting until a specific time, jobs waiting until after a certain time, or those waiting for a time before the specified time.

The −**A** option allows users to restrict the selected jobs to those that have been submitted (or altered) to charge a particular account.

The −**c** option allows users to restrict the selected jobs to those whose checkpointing interval falls within the specified range.

The −**l** option allows users to select those jobs whose resource limits fall within the range indicated by the value of the option. For example, a user could select those jobs for which the CPU time limit is greater than two hours.

The −**N** option allows users to select jobs by job name. For instance, all the parts of a task that have been divided in parallel jobs might be given the same name, and thus manipulated as a group by means of this option.

The −**q** option allows users to select jobs in a specified queue.

The −**r** option allows users to select only those jobs with a specified rerun criteria. For instance, a user might select only those jobs that can be rerun for use with the *qrerun* utility.

The −**s** option allows users to select only those jobs that are in a certain state.

The −**u** option allows users to select jobs that have been submitted to execute under a particular account.

The selection criteria provided by the options of the *qselect* utility allow users to select jobs based on all the appropriate attributes that can be assigned to jobs by the *qsub* utility.

Historically, the *qselect* utility has not been a part of existing practice; it is an improvement that has been introduced in this volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

qdel, *qrerun*, *qrls*, *qselect*, *qsub*, *touch*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

NAME

qsig — signal batch jobs

SYNOPSIS

BE

```
qsig [−s signal] job_identifier ...
```

DESCRIPTION

To signal a batch job is to send a signal to the session leader of the batch job. A batch job is signaled by sending a request to the batch server that manages the batch job. The *qsig* utility is a user-accessible batch client that requests the signaling of a batch job.

The *qsig* utility shall signal those batch jobs for which a batch *job_identifier* is presented to the utility. The *qsig* utility shall not signal any batch jobs whose batch *job_identifier*s are not presented to the utility.

The *qsig* utility shall signal batch jobs in the order in which the corresponding batch *job_identifier*s are presented to the utility. If the *qsig* utility fails to process a batch *job_identifier* successfully, the utility shall proceed to process the remaining batch *job_identifier*s, if any.

The *qsig* utility shall signal batch jobs by sending a *Signal Job Request* to the batch server that manages the batch job.

For each successfully processed batch *job_identifier*, the *qsig* utility shall have received a completion reply to each *Signal Job Request* sent to a batch server at the time the utility exits.

OPTIONS

The *qsig* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported by the implementation:

−**s** *signal* Define the signal to be sent to the batch job.

The *qsig* utility shall accept a *signal* option-argument that is either a symbolic signal name or an unsigned integer signal number (see the POSIX.1-1990 standard, Section 3.3.1.1). The *qsig* utility shall accept signal names for which the SIG prefix has been omitted.

If the *signal* option-argument is a signal name, the *qsig* utility shall send that name.

If the *signal* option-argument is a number, the *qsig* utility shall send the signal value represented by the number.

If the −**s** option is not presented to the *qsig* utility, the utility shall send the signal SIGTERM to each signaled batch job.

OPERANDS

The *qsig* utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qsig*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.

ASYNCHRONOUS EVENTS

Default.

STDOUT

An implementation of the *qsig* utility may write informative messages to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qsig* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qsig* utility waits to output the diagnostic message while attempting to locate the batch job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qsig* utility allows users to signal batch jobs.

A user may be unable to signal a batch job with the *kill* utility of the operating system for a number of reasons. First, the process ID of the batch job may be unknown to the user. Second, the processes of the batch job may be on a remote node. However, by virtue of communication between batch nodes, the *qsig* utility can arrange for the signaling of a process.

Because a batch job that is not running cannot be signaled, and because the signal may not terminate the batch job, the *qsig* utility is not a substitute for the *qdel* utility.

The options of the *qsig* utility allow the user to specify the signal that is to be sent to the batch job.

The −**s** option allows users to specify a signal by name or by number, and thus override the default signal. The POSIX.1-1990 standard defines signals by both name and number.

The *qsig* utility is a new utility, *vis-a-vis* existing practice; it has been defined in this volume of IEEE Std 1003.1-200x in response to user-perceived shortcomings in existing practice.

FUTURE DIRECTIONS

None.

SEE ALSO

kill, *qdel*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

NAME

qstat — show status of batch jobs

SYNOPSIS

BE

```
qstat [−f] job_identifier ...

qstat −Q [−f] destination ...

qstat −B [−f] server_name ...
```

DESCRIPTION

The status of a batch job, batch queue, or batch server is obtained by a request to the server. The *qstat* utility is a user-accessible batch client that requests the status of one or more batch jobs, batch queues, or servers, and writes the status information to standard output.

For each successfully processed batch *job_identifier*, the *qstat* utility shall display information about the corresponding batch job.

For each successfully processed destination, the *qstat* utility shall display information about the corresponding batch queue.

For each successfully processed server name, the *qstat* utility shall display information about the corresponding server.

The *qstat* utility shall acquire batch job status information by sending a *Job Status Request* to a batch server. The *qstat* utility shall acquire batch queue status information by sending a *Queue Status Request* to a batch server. The *qstat* utility shall acquire server status information by sending a *Server Status Request* to a batch server.

OPTIONS

The *qstat* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−f Specify that a full display is produced.

The minimum contents of a full display are specified in the STDOUT section.

Additional contents and format of a full display are implementation-defined.

−Q Specify that the operand is a destination.

The *qstat* utility shall display information about each batch queue at each destination identified as an operand.

−B Specify that the operand is a server name.

The *qstat* utility shall display information about each server identified as an operand.

OPERANDS

If the **−Q** option is presented to the *qstat* utility, the utility shall accept one or more operands that conform to the syntax for a destination (see Section 3.3.2 (on page 2325)).

If the **−B** option is presented to the *qstat* utility, the utility shall accept one or more *server_name* operands.

If neither the **−B** nor the **−Q** option is presented to the *qstat* utility, the utility shall accept one or more operands that conform to the syntax for a batch *job_identifier* (see Section 3.3.1 (on page 2324)).

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qstat*:

HOME	Determine the pathname of the user's home directory.
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_NUMERIC	Determine the locale for selecting the radix character used when writing floating-point formatted output.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If an operand presented to the *qstat* utility is a batch *job_identifier* and the −**f** option is not specified, the *qstat* utility shall display the following items on a single line, in the stated order, with white space between each item, for each successfully processed operand:

- The batch *job_identifier*
- The batch job name
- The *Job_Owner* attribute
- The CPU time used by the batch job
- The batch job state
- The batch job location

If an operand presented to the *qstat* utility is a batch *job_identifier* and the −**f** option is specified, the *qstat* utility shall display the following items for each success fully processed operand:

- The batch *job_identifier*
- The batch job name

- The *Job_Owner* attribute
- The execution user ID
- The CPU time used by the batch job
- The batch job state
- The batch job location
- Additional implementation-defined information, if any, about the batch job or batch queue

If an operand presented to the *qstat* utility is a destination, the −**Q** option is specified, and the −**f** option is not specified, the *qstat* utility shall display the following items on a single line, in the stated order, with white space between each item, for each successfully processed operand:

- The batch queue name
- The maximum number of batch jobs that shall be run in the batch queue concurrently
- The total number of batch jobs in the batch queue
- The status of the batch queue
- For each state, the number of batch jobs in that state in the batch queue and the name of the state
- The type of batch queue (execution or routing)

If the operands presented to the *qstat* utility are destinations, the −**Q** option is specified, and the −**f** option is specified, the *qstat* utility shall display the following items for each successfully processed operand:

- The batch queue name
- The maximum number of batch jobs that shall be run in the batch queue concurrently
- The total number of batch jobs in the batch queue
- The status of the batch queue
- For each state, the number of batch jobs in that state in the batch queue and the name of the state
- The type of batch queue (execution or routing)
- Additional implementation-defined information, if any, about the batch queue

If the operands presented to the *qstat* utility are batch server names, the −**B** option is specified, and the −**f** option is not specified, the *qstat* utility shall display the following items on a single line, in the stated order, with white space between each item, for each successfully processed operand:

- The batch server name
- The maximum number of batch jobs that shall be run in the batch queue concurrently
- The total number of batch jobs managed by the batch server
- The status of the batch server
- For each state, the number of batch jobs in that state and the name of the state

If the operands presented to the *qstat* utility are server names, the −**B** option is specified, and the −**f** option is specified, the *qstat* utility shall display the following items for each successfully processed operand:

- The server name
- The maximum number of batch jobs that shall be run in the batch queue concurrently
- The total number of batch jobs managed by the server
- The status of the server
- For each state, the number of batch jobs in that state and the name of the state
- Additional implementation-defined information, if any, about the server

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

In addition to the default behavior, the *qstat* utility shall not be required to write a diagnostic message to standard error when the error reply received from a batch server indicates that the batch *job_identifier* does not exist on the server. Whether or not the *qstat* utility waits to output the diagnostic message while attempting to locate the batch job on other servers is implementation-defined.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qstat* utility allows users to display the status of jobs and listing the batch jobs in queues.

The operands of the *qstat* utility may be either job identifiers, queues (specified as destination identifiers), or batch server names. The **–Q** and **–B** options, or absence thereof, indicate the nature of the operands.

The other options of the *qstat* utility allow the user to control the amount of information displayed and the format in which it is displayed. Should a user wish to display the status of a set of jobs that match a selection criteria, the *qselect* utility may be used to acquire such a list.

The **–f** option allows users to request a ‘‘full’’ display in an implementation-defined format.

Historically, the *qstat* utility has been a part of the NQS and its derivatives, the existing practice on which it is based.

FUTURE DIRECTIONS

None.

SEE ALSO

qselect, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

IEEE PASC Interpretation 1003.2 #191 is applied, removing the following ENVIRONMENT VARIABLES listed as affecting *qstat*: *COLUMNS*, *LINES*, *LOGNAME*, *TERM*, and *TZ*.

The *LC_TIME* entry is also removed from the ENVIRONMENT VARIABLES section.

NAME

qsub — submit a script

SYNOPSIS

BE
```
qsub [−a date_time][−A account_string][−c interval]
    [−C directive_prefix][−e path_name][−h][−j join_list][−k keep_list]
    [−m mail_options][−M mail_list][−N name]
    [−o path_name][−p priority][−q destination][−r y|n]
    [−S path_name_list][−u user_list][−v variable_list][−V]
    [−z][script]
```

DESCRIPTION

To submit a script is to create a batch job that executes the script. A script is submitted by a request to a batch server. The *qsub* utility is a user-accessible batch client that submits a script.

Upon successful completion, the *qsub* utility shall have created a batch job that will execute the submitted script.

The *qsub* utility shall submit a script by sending a *Queue Job Request* to a batch server.

The *qsub* utility shall place the value of the following environment variables in the *Variable_List* attribute of the batch job: *HOME*, *LANG*, *LOGNAME*, *PATH*, *MAIL*, *SHELL*, and *TZ*. The name of the environment variable shall be the current name prefixed with the string PBS_O_.

Note: If the current value of the *HOME* variable in the environment space of the *qsub* utility is **/aa/bb/cc**, then *qsub* shall place *PBS_O_HOME*=**/aa/bb/cc** in the *Variable_List* attribute of the batch job.

In addition to the variables described above, the *qsub* utility shall add the following variables with the indicated values to the variable list:

PBS_O_WORKDIR	The absolute path of the current working directory of the *qsub* utility process.
PBS_O_HOST	The name of the host on which the *qsub* utility is running.

OPTIONS

The *qsub* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported by the implementation:

−**a** *date_time* Define the time at which a batch job becomes eligible for execution.

The *qsub* utility shall accept an option-argument that conforms to the syntax of the *time* operand of the *touch* utility.

Table 4-18 Environment Variable Values (Utilities)

Variable Name	Value at qsub Time
PBS_O_HOME	*HOME*
PBS_O_HOST	Client host name
PBS_O_LANG	*LANG*
PBS_O_LOGNAME	*LOGNAME*
PBS_O_PATH	*PATH*
PBS_O_MAIL	*MAIL*
PBS_O_SHELL	*SHELL*
PBS_O_TZ	*TZ*
PBS_O_WORKDIR	Current working directory

Note: The server that initiates execution of the batch job will add other variables to the batch job's environment; see Section 3.2.2.1 (on page 2308).

The *qsub* utility shall set the *Execution_Time* attribute of the batch job to the number of seconds since the Epoch that is equivalent to the local time expressed by the value of the *date_time* option-argument. The Epoch is defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.149, Epoch.

If the −**a** option is not presented to the *qsub* utility, the utility shall set the *Execution_Time* attribute of the batch job to a time (number of seconds since the Epoch) that is earlier than the time at which the utility exits.

−**A** *account_string*

Define the account to which the resource consumption of the batch job should be charged.

The syntax of the *account_string* option-argument is unspecified.

The *qsub* utility shall set the *Account_Name* attribute of the batch job to the value of the *account_string* option-argument.

If the −**A** option is not presented to the *qsub* utility, the utility shall omit the *Account_Name* attribute from the attributes of the batch job.

−**c** *interval* Define whether the batch job should be checkpointed, and if so, how often.

The *qsub* utility shall accept a value for the interval option-argument that is one of the following:

`n` No checkpointing shall be performed on the batch batch job (NO_CHECKPOINT).

`s` Checkpointing shall be performed only when the batch server is shut down (CHECKPOINT_AT_SHUTDOWN).

`c` Automatic periodic checkpointing shall be performed at the *Minimum_Cpu_Interval* attribute of the batch queue, in units of CPU minutes (CHECKPOINT_AT_MIN_CPU_INTERVAL).

`c=`*minutes* Automatic periodic checkpointing shall be performed every minutes of CPU time, or every *Minimum_Cpu_Interval* minutes, whichever is greater. The *minutes* argument shall conform to the syntax for unsigned integers and shall be greater than zero.

The *qsub* utility shall set the *Checkpoint* attribute of the batch job to the value of the *interval* option-argument.

If the **–c** option is not presented to the *qsub* utility, the utility shall set the *Checkpoint* attribute of the batch job to the single character `'u'` (CHECKPOINT_UNSPECIFIED).

–C *directive_prefix*
Define the prefix that declares a directive to the *qsub* utility within the script.

The *directive_prefix* is not a batch job attribute; it affects the behavior of the *qsub* utility.

If the **–C** option is presented to the *qsub* utility, and the value of the *directive_prefix* option-argument is the null string, the utility shall not scan the script file for directives. If the **–C** option is not presented to the *qsub* utility, then the value of the *PBS_DPREFIX* environment variable is used. If the environment variable is not defined, then #PBS encoded in the portable character set is the default.

–e *path_name* Define the path to be used for the standard error stream of the batch job.

The *qsub* utility shall accept a *path_name* option-argument which can be preceded by a host name element of the form *hostname*:.

If the *path_name* option-argument constitutes an absolute pathname, the *qsub* utility shall set the *Error_Path* attribute of the batch job to the value of the *path_name* option-argument.

If the *path_name* option-argument constitutes a relative pathname and no host name element is specified, the *qsub* utility shall set the *Error_Path* attribute of the batch job to the value of the absolute pathname derived by expanding the *path_name* option-argument relative to the current directory of the process executing *qsub*.

If the *path_name* option-argument constitutes a relative pathname and a host name element is specified, the *qsub* utility shall set the *Error_Path* attribute of the batch job to the value of the *path_name* option-argument without expansion. The host name element shall be included.

If the *path_name* option-argument does not include a host name element, the *qsub* utility shall prefix the pathname with *hostname*:, where *hostname* is the name of the host upon which the *qsub* utility is being executed.

If the **–e** option is not presented to the *qsub* utility, the utility shall set the *Error_Path* attribute of the batch job to the host name and path of the current directory of the submitting process and the default filename.

The default filename for standard error has the following format:

`job_name.esequence_number`

–h Specify that a USER hold is applied to the batch job.

The *qsub* utility shall set the value of the *Hold_Types* attribute of the batch job to the value USER.

If the **–h** option is not presented to the *qsub* utility, the utility shall set the *Hold_Types* attribute of the batch job to the value NO_HOLD.

–j *join_list* Define which streams of the batch job are to be merged. The *qsub* **–j** option shall accept a value for the *join_list* option-argument that is a string of alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

The *qsub* utility shall accept a *join_list* option-argument that consists of one or more of the characters `'e'` and `'o'` or the single character `'n'`.

All of the other batch job output streams specified will be merged into the output stream represented by the character listed first in the *join_list* option-argument.

For each unique character in the *join_list* option-argument, the *qsub* utility shall add a value to the *Join_Path* attribute of the batch job as follows, each representing a different batch job stream to join:

`e` The standard error of the batch batch job (JOIN_STD_ERROR).

`o` The standard output of the batch batch job (JOIN_STD_OUTPUT).

An existing *Join_Path* attribute can be cleared by the following join type:

`n` NO_JOIN

If `'n'` is specified, then no files are joined. The *qsub* utility shall consider it an error if any join type other than `'n'` is combined with join type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'e'`, `'o'`, or `'n'` within the *join_list* option-argument. The *qsub* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters.

An implementation may define other join types. The conformance document for an implementation shall describe any additional batch job streams, how they are specified, their internal behavior, and how they affect the behavior of the utility.

If the −**j** option is not presented to the *qsub* utility, the utility shall set the value of the *Join_Path* attribute of the batch job to NO_JOIN.

−**k** *keep_list* Define which output of the batch job to retain on the execution host.

The *qsub* −**k** option shall accept a value for the *keep_list* option-argument that is a string of alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

The *qsub* utility shall accept a *keep_list* option-argument that consists of one or more of the characters `'e'` and `'o'` or the single character `'n'`.

For each unique character in the *keep_list* option-argument, the *qsub* utility shall add a value to the *Keep_Files* attribute of the batch job as follows, each representing a different batch job stream to keep:

`e` The standard error of the batch batch job (KEEP_STD_ERROR).

`o` The standard output of the batch batch job (KEEP_STD_OUTPUT).

If both `'e'` and `'o'` are specified, then both files are retained. An existing *Keep_Files* attribute can be cleared by the following keep type:

`n` NO_KEEP

If `'n'` is specified, then no files are retained. The *qsub* utility shall consider it an error if any keep type other than `'n'` is combined with keep type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'e'`, `'o'`, or `'n'` within the *keep_list* option-argument. The *qsub* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters.

An implementation may define other keep types. The conformance document for an implementation shall describe any additional keep types, how they are specified, their internal behavior, and how they affect the behavior of the utility. If the −**k** option is not presented to the *qsub* utility, the utility shall set the *Keep_Files* attribute of the batch job to the value NO_KEEP.

−**m** *mail_options*

Define the points in the execution of the batch job at which the batch server that manages the batch job shall send mail about a change in the state of the batch job.

The *qsub* −**m** option shall accept a value for the *mail_options* option-argument that is a string of alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set).

The *qsub* utility shall accept a value for the *mail_options* option-argument that is a string of one or more of the characters `'e'`, `'b'`, and `'a'`, or the single character `'n'`.

For each unique character in the *mail_options* option-argument, the *qsub* utility shall add a value to the *Mail_Users* attribute of the batch job as follows, each representing a different time during the life of a batch job at which to send mail:

`e`	MAIL_AT_EXIT
`b`	MAIL_AT_BEGINNING
`a`	MAIL_AT_ABORT

If any of these characters are duplicated in the *mail_options* option-argument, the duplicates shall be ignored.

An existing *Mail_Points* attribute can be cleared by the following mail type:

`n`	NO_MAIL

If `'n'` is specified, then mail is not sent. The *qsub* utility shall consider it an error if any mail type other than `'n'` is combined with mail type `'n'`.

Strictly conforming applications shall not repeat any of the characters `'e'`, `'b'`, `'a'`, or `'n'` within the *mail_options* option-argument.

The *qsub* utility shall permit the repetition of characters, but shall not assign additional meaning to the repeated characters. An implementation may define other mail types. The conformance document for an implementation shall describe any additional mail types, how they are specified, their internal behavior, and how they affect the behavior of the utility.

If the −**m** option is not presented to the *qsub* utility, the utility shall set the *Mail_Points* attribute to the value MAIL_AT_ABORT.

−**M** *mail_list* Define the list of users to which a batch server that executes the batch job shall send mail, if the server sends mail about the batch job.

The syntax of the *mail_list* option-argument is unspecified.

If the implementation of the *qsub* utility uses a name service to locate users, the utility should accept the syntax used by the name service.

If the implementation of the *qsub* utility does not use a name service to locate users, the implementation should accept the following syntax for user names:

`mail_address[,,mail_address,, ...]`

The interpretation of *mail_address* is implementation-defined.

The *qsub* utility shall set the *Mail_Users* attribute of the batch job to the value of the *mail_list* option-argument.

If the −**M** option is not presented to the *qsub* utility, the utility shall place only the user name and host name for the current process in the *Mail_Users* attribute of the batch job.

−**N** *name* Define the name of the batch job.

The *qsub* −**N** option shall accept a value for the *name* option-argument that is a string of up to 15 alphanumeric characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set) where the first character is alphabetic.

The *qsub* utility shall set the value of the *Job_Name* attribute of the batch job to the value of the *name* option-argument.

If the −**N** option is not presented to the *qsub* utility, the utility shall set the *Job_Name* attribute of the batch job to the name of the *script* argument from which the directory specification if any, has been removed.

If the −**N** option is not presented to the *qsub* utility, and the script is read from standard input, the utility shall set the *Job_Name* attribute of the batch job to the value STDIN.

−**o** *path_name* Define the path for the standard output of the batch job.

The *qsub* utility shall accept a *path_name* option-argument that conforms to the syntax of the *path_name* element defined in the System Interfaces volume of IEEE Std 1003.1-200x, which can be preceded by a host name element of the form *hostname*:.

If the *path_name* option-argument constitutes an absolute pathname, the *qsub* utility shall set the *Output_Path* attribute of the batch job to the value of the *path_name* option-argument without expansion.

If the *path_name* option-argument constitutes a relative pathname and no host name element is specified, the *qsub* utility shall set the *Output_Path* attribute of the batch job to the pathname derived by expanding the value of the *path_name* option-argument relative to the current directory of the process executing the *qsub*.

If the *path_name* option-argument constitutes a relative pathname and a host name element is specified, the *qsub* utility shall set the *Output_Path* attribute of the batch job to the value of the *path_name* option-argument without expansion.

If the *path_name* option-argument does not specify a host name element, the *qsub* utility shall prefix the pathname with *hostname*:, where *hostname* is the name of the host upon which the *qsub* utility is executing.

If the −**o** option is not presented to the *qsub* utility, the utility shall set the *Output_Path* attribute of the batch job to the host name and path of the current directory of the submitting process and the default filename.

The default filename for standard output has the following format:

`job_name.osequence_number`

−p *priority* Define the priority the batch job should have relative to other batch jobs owned by the batch server.

The *qsub* utility shall set the *Priority* attribute of the batch job to the value of the *priority* option-argument.

If the **−p** option is not presented to the *qsub* utility, the value of the *Priority* attribute is implementation-defined.

The *qsub* utility shall accept a value for the *priority* option-argument that conforms to the syntax for signed decimal integers, and which is not less than −1 024 and not greater than 1 023.

−q *destination*

Define the destination of the batch job.

The destination is not a batch job attribute; it determines the batch server, and possibly the batch queue, to which the *qsub* utility batch queues the batch job.

The *qsub* utility shall submit the script to the batch server named by the *destination* option-argument or the server that owns the batch queue named in the *destination* option-argument.

The *qsub* utility shall accept an option-argument for the **−q** option that conforms to the syntax for a destination (see Section 3.3.2 (on page 2325)).

If the **−q** option is not presented to the *qsub* utility, the *qsub* utility shall submit the batch job to the default destination. The mechanism for determining the default destination is implementation-defined.

−r *y* | *n* Define whether the batch job is rerunable.

If the value of the option-argument is *y*, the *qsub* utility shall set the *Rerunable* attribute of the batch job to TRUE.

If the value of the option-argument is *n*, the *qsub* utility shall set the *Rerunable* attribute of the batch job to FALSE.

If the **−r** option is not presented to the *qsub* utility, the utility shall set the *Rerunable* attribute of the batch job to TRUE.

−S *path_name_list*

Define the pathname to the shell under which the batch job is to execute.

The *qsub* utility shall accept a *path_name_list* option-argument that conforms to the following syntax:

```
pathname[@host][,,pathname[@host],, ...]
```

The *qsub* utility shall allow only one pathname for a given host name. The *qsub* utility shall allow only one pathname that is missing a corresponding host name.

The *qsub* utility shall add a value to the *Shell_Path_List* attribute of the batch job for each entry in the *path_name_list* option-argument.

If the **−S** option is not presented to the *qsub* utility, the utility shall set the *Shell_Path_List* attribute of the batch job to the null string.

The conformance document for an implementation shall describe the mechanism used to set the default shell and determine the current value of the default shell. An implementation shall provide a means for the installation to set the default shell to the login shell of the user under which the batch job is to execute. See

Section 3.3.3 (on page 2325) for a means of removing *keyword*=*value* (and *value*@*keyword*) pairs and other general rules for list-oriented batch job attributes.

−u *user_list* Define the user name under which the batch job is to execute.

The *qsub* utility shall accept a *user_list* option-argument that conforms to the following syntax:

`username[@host][,,username[@host],, ...]`

The *qsub* utility shall accept only one user name that is missing a corresponding host name. The *qsub* utility shall accept only one user name per named host.

The *qsub* utility shall add a value to the *User_List* attribute of the batch job for each entry in the *user_list* option-argument.

If the **−u** option is not presented to the *qsub* utility, the utility shall set the *User_List* attribute of the batch job to the user name from which the utility is executing. See Section 3.3.3 (on page 2325) for a means of removing *keyword*=*value* (and *value*@*keyword*) pairs and other general rules for list-oriented batch job attributes.

−v *variable_list*

Add to the list of variables that are exported to the session leader of the batch job.

A *variable_list* is a set of strings of either the form <*variable*> or <*variable*=*value*>, delimited by commas.

If the **−v** option is presented to the *qsub* utility, the utility shall also add, to the environment *Variable_List* attribute of the batch job, every variable named in the environment *variable_list* option-argument and, optionally, values of specified variables.

If a value is not provided on the command line, the *qsub* utility shall set the value of each variable in the environment *Variable_List* attribute of the batch job to the value of the corresponding environment variable for the process in which the utility is executing; see Table 4-18 (on page 3003).

A conforming application shall not repeat a variable in the environment *variable_list* option-argument.

The *qsub* utility shall not repeat a variable in the environment *Variable_List* attribute of the batch job. See Section 3.3.3 (on page 2325) for a means of removing *keyword*=*value* (and *value*@*keyword*) pairs and other general rules for list-oriented batch job attributes.

−V Specify that all of the environment variables of the process are exported to the context of the batch job.

The *qsub* utility shall place every environment variable in the process in which the utility is executing in the list and shall set the value of each variable in the attribute to the value of that variable in the process.

−z Specify that the utility does not write the batch *job_identifier* of the created batch job to standard output.

If the **−z** option is presented to the *qsub* utility, the utility shall not write the batch *job_identifier* of the created batch job to standard output.

If the **−z** option is not presented to the *qsub* utility, the utility shall write the identifier of the created batch job to standard output.

OPERANDS

The *qsub* utility shall accept a *script* operand that indicates the path to the script of the batch job.

If the *script* operand is not presented to the *qsub* utility, or if the operand is the single-character string `'-'`, the utility shall read the script from standard input.

If the script represents a partial path, the *qsub* utility shall expand the path relative to the current directory of the process executing the utility.

STDIN

The *qsub* utility reads the script of the batch job from standard input if the script operand is omitted or is the single character `'-'`.

INPUT FILES

In addition to binding the file indicated by the *script* operand to the batch job, the *qsub* utility reads the script file and acts on directives in the script.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *qsub*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LOGNAME	Determine the login name of the user.
PBS_DPREFIX	Determine the default prefix for directives within the script.
SHELL	Determine the pathname of the preferred command language interpreter of the user.
TZ	Determine the timezone used to interpret the *date-time* option-argument. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Once created, a batch job exists until it exits, aborts, or is deleted.

After a batch job is created by the *qsub* utility, batch servers might route, execute, modify, or delete the batch job.

STDOUT

The *qsub* utility writes the batch *job_identifier* assigned to the batch job to standard output, unless the **−z** option is specified.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

Script Preservation

The *qsub* utility shall make the script available to the server executing the batch job in such a way that the server executes the script as it exists at the time of submission.

The *qsub* utility can send a copy of the script to the server with the *Queue Job Request* or store a temporary copy of the script in a location specified to the server.

Option Specification

A script can contain directives to the *qsub* utility.

The *qsub* utility shall scan the lines of the script for directives, skipping blank lines, until the first line that begins with a string other than the directive string; if directives occur on subsequent lines, the utility shall ignore those directives.

Lines are separated by a <newline>. If the first line of the script begins with `"#!"` or a colon (`':'`), then it is skipped. The *qsub* utility shall process a line in the script as a directive if and only if the string of characters from the first non-white-space character on the line until the first <space> or <tab> on the line match the directive prefix. If a line in the script contains a directive and the final characters of the line are backslash (`'\'`) and <newline>, then the next line shall be interpreted as a continuation of that directive.

The *qsub* utility shall process the options and option-arguments contained on the directive prefix line using the same syntax as if the options were input on the *qsub* utility.

The *qsub* utility shall continue to process a directive prefix line until after a <newline> is encountered. An implementation may ignore lines which, according to the syntax of the shell that will interpret the script, are comments. An implementation shall describe in the conformance document the format of any shell comments that it will recognize.

If an option is present in both a directive and the arguments to the *qsub* utility, the utility shall ignore the option and the corresponding option-argument, if any, in the directive.

If an option that is present in the directive is not present in the arguments to the *qsub* utility, the utility shall process the option and the option-argument, if any.

In order of preference, the *qsub* utility shall select the directive prefix from one of the following sources:

- If the **–C** option is presented to the utility, the value of the *directive_prefix* option-argument
- If the environment variable *PBS_DPREFIX* is defined, the value of that variable
- The four-character string `"#PBS"` encoded in the portable character set

If the **–C** option is present in the script file it shall be ignored.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The *qsub* utility allows users to create a batch job that will process the script specified as the operand of the utility.

The options of the *qsub* utility allow users to control many aspects of the queuing and execution of a batch job.

The −**a** option allows users to designate the time after which the batch job will become eligible to run. By specifying an execution time, users can take advantage of resources at off-peak hours, synchronize jobs with chronologically predictable events, and perhaps take advantage of off-peak pricing of computing time. For these reasons and others, a timing option is existing practice on the part of almost every batch system, including NQS.

The −**A** option allows users to specify the account that will be charged for the batch job. Support for account is not mandatory for conforming batch servers.

The −**C** option allows users to prescribe the prefix for directives within the script file. The default prefix `"#PBS"` may be inappropriate if the script will be interpreted with an alternate shell, as specified by the −**S** option.

The −**c** option allows users to establish the checkpointing interval for their jobs. A checkpointing system, which is not defined by this volume of IEEE Std 1003.1-200x, allows recovery of a batch job at the most recent checkpoint in the event of a crash. Checkpointing is typically used for jobs that consume expensive computing time or must meet a critical schedule. Users should be allowed to make the tradeoff between the overhead of checkpointing and the risk to the timely completion of the batch job; therefore, this volume of IEEE Std 1003.1-200x provides the checkpointing interval option. Support for checkpointing is optional for batch servers.

The −**e** option allows users to redirect the standard error streams of their jobs to a non-default path. For example, if the submitted script generally produces a great deal of useless error output, a user might redirect the standard error output to the null device. Or, if the file system holding the default location (the home directory of the user) has too little free space, the user might redirect the standard error stream to a file in another file system.

The −**h** option allows users to create a batch job that is held until explicitly released. The ability to create a held job is useful when some external event must complete before the batch job can execute. For example, the user might submit a held job and release it when the system load has dropped.

The −**j** option allows users to merge the standard error of a batch job into its standard output stream, which has the advantage of showing the sequential relationship between output and error messages.

The −**m** option allows users to designate those points in the execution of a batch job at which mail will be sent to the submitting user, or to the account(s) indicated by the −**M** option. By requesting mail notification at points of interest in the life of a job, the submitting user, or other designated users, can track the progress of a batch job.

The –**N** option allows users to associate a name with the batch job. The job name in no way affects the processing of the batch job, but rather serves as a mnemonic handle for users. For example, the batch job name can help the user distinguish between multiple jobs listed by the *qstat* utility.

The –**o** option allows users to redirect the standard output stream. A user might, for example, wish to redirect to the null device the standard output stream of a job that produces copious yet superfluous output.

The –**P** option allows users to designate the relative priority of a batch job for selection from a queue.

The –**q** option allows users to specify an initial queue for the batch job. If the user specifies a routing queue, the batch batch server routes the batch job to another queue for execution or further routing. If the user specifies a non-routing queue, the batch server of the queue eventually executes the batch job.

The –**r** option allows users to control whether the submitted job will be rerun if the controlling batch node fails during execution of the batch job. The –**r** option likewise allows users to indicate whether or not the batch job is eligible to be rerun by the *qrerun* utility. Some jobs cannot be correctly rerun because of changes they make in the state of databases or other aspects of their environment. This volume of IEEE Std 1003.1-200x specifies that the default, if the –**r** option is not presented to the utility, will be that the batch job cannot be rerun, since the result of rerunning a non-rerunable job might be catastrophic.

The –**S** option allows users to specify the program (usually a shell) that will be invoked to process the script of the batch job. This option has been modified to allow a list of shell names and locations associated with different hosts.

The –**u** option is useful when the submitting user is authorized to use more than one account on a given host, in which case the –**u** option allows the user to select from among those accounts. The option-argument is a list of user-host pairs, so that the submitting user can provide different user identifiers for different nodes in the event the batch job is routed. The –**u** option provides a lot of flexibility to accommodate sites with complex account structures. Users that have the same user identifier on all the hosts they are authorized to use will not need to use the –**u** option.

The –**V** option allows users to export all their current environment variables, as of the time the batch job is submitted, to the context of the processes of the batch job.

The –**v** option allows users to export specific environment variables from their current process to the processes of the batch job.

The –**z** option allows users to suppress the writing of the batch job identifier to standard output. The –**z** option is an existing NQS practice that has been standardized.

Historically, the *qsub* utility has served the batch job-submission function in the NQS system, the existing practice on which it is based. Some changes and additions have been made to the *qsub* utility in this volume of IEEE Std 1003.1-200x, *vis-a-vis* NQS, as a result of the growing pool of experience with distributed batch systems.

The set of features of the *qsub* utility as defined in this volume of IEEE Std 1003.1-200x appears to incorporate all the common existing practice on potentially POSIX-conformant platforms.

FUTURE DIRECTIONS

None.

SEE ALSO

qrerun, *qstat*, *touch*, Chapter 3 (on page 2303)

CHANGE HISTORY

Derived from IEEE Std 1003.2d-1994.

Issue 6

The **–l** option has been removed as there is no portable description of the resources that are allowed or required by the batch job.

NAME

read — read a line from standard input

SYNOPSIS

```
read [−r] var...
```

DESCRIPTION

The *read* utility shall read a single line from standard input.

By default, unless the **−r** option is specified, backslash (`'\'`) shall act as an escape character, as described in Section 2.2.1 (on page 2232). If standard input is a terminal device and the invoking shell is interactive, *read* shall prompt for a continuation line when:

- The shell reads an input line ending with a backslash, unless the **−r** option is specified.
- A here-document is not terminated after a <newline> is entered.

The line shall be split into fields as in the shell (see Section 2.6.5 (on page 2243)); the first field shall be assigned to the first variable *var*, the second field to the second variable *var*, and so on. If there are fewer *var* operands specified than there are fields, the leftover fields and their intervening separators shall be assigned to the last *var*. If there are fewer fields than *var*s, the remaining *var*s shall be set to empty strings.

The setting of variables specified by the *var* operands shall affect the current shell execution environment; see Section 2.12 (on page 2263). If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(read foo)
nohup read ...
find . −exec read ... \;
```

it shall not affect the shell variables in the caller's environment.

OPTIONS

The *read* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option is supported:

−r	Do not treat a backslash character in any special way. Consider each backslash to be part of the input line.

OPERANDS

The following operand shall be supported:

var	The name of an existing or nonexisting shell variable.

STDIN

The standard input shall be a text file.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *read*:

IFS	Determine the internal field separators used to delimit fields; see Section 2.5.3 (on page 2236).
LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables

	used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
PS2	Provide the prompt string that an interactive shell shall write to standard error when a line ending with a backslash is read and the **–r** option was not specified, or if a here-document is not terminated after a <newline> is entered.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used for diagnostic messages and prompts for continued input.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	End-of-file was detected or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The **–r** option is included to enable *read* to subsume the purpose of the *line* utility, which is not included in IEEE Std 1003.1-200x.

The results are undefined if an end-of-file is detected following a backslash at the end of a line when **–r** is not specified.

EXAMPLES

The following command:

```
while read –r xx yy
do
    printf "%s %s\n" "$yy" "$xx"
done < input_file
```

prints a file with the first field of each line moved to the end of the line.

RATIONALE

The *read* utility historically has been a shell built-in. It was separated off into its own utility to take advantage of the richer description of functionality introduced by this volume of IEEE Std 1003.1-200x.

Since *read* affects the current shell execution environment, it is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(read foo)
nohup read ...
find . −exec read ... \;
```

it does not affect the shell variables in the environment of the caller.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

NAME

renice — set nice values of running processes

SYNOPSIS

UP

```
renice -n increment [-g | -p | -u] ID ...
```

DESCRIPTION

The *renice* utility shall request that the nice values (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.239, Nice Value) of one or more running processes be changed. By default, the applicable processes are specified by their process IDs. When a process group is specified (see **−g**), the request shall apply to all processes in the process group.

The nice value shall be bounded in an implementation-defined manner. If the requested *increment* would raise or lower the nice value of the executed utility beyond implementation-defined limits, then the limit whose value was exceeded shall be used.

When a user is *renice*d, the request applies to all processes whose saved set-user-ID matches the user ID corresponding to the user.

Regardless of which options are supplied or any other factor, *renice* shall not alter the nice values of any process unless the user requesting such a change has appropriate privileges to do so for the specified process. If the user lacks appropriate privileges to perform the requested action, the utility shall return an error status.

The saved set-user-ID of the user's process shall be checked instead of its effective user ID when *renice* attempts to determine the user ID of the process in order to determine whether the user has appropriate privileges.

OPTIONS

The *renice* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−g	Interpret all operands as unsigned decimal integer process group IDs.
−n *increment*	Specify how the nice value of the specified process or processes is to be adjusted. The *increment* option-argument is a positive or negative decimal integer that shall be used to modify the nice value of the specified process or processes. Positive *increment* values shall cause a lower nice value. Negative *increment* values may require appropriate privileges and shall cause a higher nice value.
−p	Interpret all operands as unsigned decimal integer process IDs. The **−p** option is the default if no options are specified.
−u	Interpret all operands as users. If a user exists with a user name equal to the operand, then the user ID of that user is used in further processing. Otherwise, if the operand represents an unsigned decimal integer, it shall be used as the numeric user ID of the user.

OPERANDS

The following operands shall be supported:

ID	A process ID, process group ID, or user name/user ID, depending on the option selected.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *renice*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. Adjust the nice value so that process IDs 987 and 32 would have a lower nice value:

   ```
   renice −n 5 −p 987 32
   ```

2. Adjust the nice value so that group IDs 324 and 76 would have a higher nice value, if the user has the appropriate privileges to do so:

   ```
   renice −n −4 −g 324 76
   ```

3. Adjust the nice value so that numeric user ID 8 and user **sas** would have a lower nice value:

   ```
   renice −n 4 −u 8 sas
   ```

Useful nice value increments on historical systems include 19 or 20 (the affected processes run only when nothing else in the system attempts to run) and any negative number (to make processes run faster).

RATIONALE

The *gid*, *pid*, and *user* specifications do not fit either the definition of operand or option-argument. However, for clarity, they have been included in the OPTIONS section, rather than the OPERANDS section.

The definition of nice value is not intended to suggest that all processes in a system have priorities that are comparable. Scheduling policy extensions such as the realtime priorities in the System Interfaces volume of IEEE Std 1003.1-200x make the notion of a single underlying priority for all scheduling policies problematic. Some implementations may implement the *nice*-related features to affect all processes on the system, others to affect just the general time-sharing activities implied by this volume of IEEE Std 1003.1-200x, and others may have no effect at all. Because of the use of ‘‘implementation-defined’’ in *nice* and *renice*, a wide range of implementation strategies are possible.

Originally, this utility was written in the historical manner, using the term ‘‘nice value’’. This was always a point of concern with users because it was never intuitively obvious what this meant. With a newer version of *renice*, which used the term ‘‘system scheduling priority’’, it was hoped that novice users could better understand what this utility was meant to do. Also, it would be easier to document what the utility was meant to do. Unfortunately, the addition of the POSIX realtime scheduling capabilities introduced the concepts of process and thread scheduling priorities that were totally unaffected by the *nice*/*renice* utilities or the *nice*( )/*setpriority*( ) functions. Continuing to use the term ‘‘system scheduling priority’’ would have incorrectly suggested that these utilities and functions were indeed affecting these realtime priorities. It was decided to revert to the historical term ‘‘nice value’’ to reference this unrelated process attribute.

Although this utility has use by system administrators (and in fact appears in the system administration portion of the BSD documentation), the standard developers considered that it was very useful for individual end users to control their own processes.

FUTURE DIRECTIONS

None.

SEE ALSO

nice

CHANGE HISTORY

First released in Issue 4.

Issue 5

In the SYNOPSIS, an ellipsis is added to the **–u** option in all three obsolescent forms.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

The obsolescent forms of the SYNOPSIS are removed.

Text previously conditional on POSIX_SAVED_IDS is mandatory in this issue. This is a FIPS requirement.

NAME

rm — remove directory entries

SYNOPSIS

```
rm [−fiRr] file...
```

DESCRIPTION

The *rm* utility shall remove the directory entry specified by each *file* argument.

If either of the files dot or dot-dot are specified as the basename portion of an operand (that is, the final pathname component), *rm* shall write a diagnostic message to standard error and do nothing more with such operands.

For each *file* the following steps shall be taken:

1. If the *file* does not exist:
   a. If the **−f** option is not specified, *rm* shall write a diagnostic message to standard error.
   b. Go on to any remaining *files*.
2. If *file* is of type directory, the following steps shall be taken:
   a. If neither the **−R** option nor the **−r** option is specified, *rm* shall write a diagnostic message to standard error, do nothing more with *file*, and go on to any remaining files.
   b. If the **−f** option is not specified, and either the permissions of *file* do not permit writing and the standard input is a terminal or the **−i** option is specified, *rm* shall write a prompt to standard error and read a line from the standard input. If the response is not affirmative, *rm* shall do nothing more with the current file and go on to any remaining files.
   c. For each entry contained in *file*, other than dot or dot-dot, the four steps listed here (1 to 4) shall be taken with the entry as if it were a *file* operand. The *rm* utility shall not traverse directories by following symbolic links into other parts of the hierarchy, but shall remove the links themselves.
   d. If the **−i** option is specified, *rm* shall write a prompt to standard error and read a line from the standard input. If the response is not affirmative, *rm* shall do nothing more with the current file, and go on to any remaining files.
3. If *file* is not of type directory, the **−f** option is not specified, and either the permissions of *file* do not permit writing and the standard input is a terminal or the **−i** option is specified, *rm* shall write a prompt to the standard error and read a line from the standard input. If the response is not affirmative, *rm* shall do nothing more with the current file and go on to any remaining files.
4. If the current file is a directory, *rm* shall perform actions equivalent to the *rmdir*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x called with a pathname of the current file used as the *path* argument. If the current file is not a directory, *rm* shall perform actions equivalent to the *unlink*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x called with a pathname of the current file used as the *path* argument.

   If this fails for any reason, *rm* shall write a diagnostic message to standard error, do nothing more with the current file, and go on to any remaining files.

The *rm* utility shall be able to descend to arbitrary depths in a file hierarchy, and shall not fail due to path length limitations (unless an operand specified by the user exceeds system limitations).

OPTIONS

The *rm* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

–f	Do not prompt for confirmation. Do not write diagnostic messages or modify the exit status in the case of nonexistent operands. Any previous occurrences of the **–i** option shall be ignored.
–i	Prompt for confirmation as described previously. Any previous occurrences of the **–f** option shall be ignored.
–R	Remove file hierarchies. See the DESCRIPTION.
–r	Equivalent to **–R**.

OPERANDS

The following operand shall be supported:

file	A pathname of a directory entry to be removed.

STDIN

The standard input shall be used to read an input line in response to each prompt specified in the STDOUT section. Otherwise, the standard input shall not be used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *rm*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments) and the behavior of character classes within regular expressions used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_MESSAGES	Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Prompts shall be written to standard error under the conditions specified in the DESCRIPTION and OPTIONS sections. The prompts shall contain the *file* pathname, but their format is otherwise unspecified. The standard error also shall be used for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All of the named directory entries for which *rm* performed actions equivalent to *rmdir*( ) or *unlink*( ) functions were removed.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *rm* utility is forbidden to remove the names dot and dot-dot in order to avoid the consequences of inadvertently doing something like:

```
rm −r .*
```

Some implementations do not permit the removal of the last link to an executable binary file that is being executed; see the [EBUSY] error in the *unlink*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. Thus, the *rm* utility can fail to remove such files. |

The −**i** option causes *rm* to prompt and read the standard input even if the standard input is not a terminal, but in the absence of −**i** the mode prompting is not done when the standard input is not a terminal.

EXAMPLES

1. The following command:

   ```
   rm a.out core
   ```

   removes the directory entries: **a.out** and **core**.

2. The following command:

   ```
   rm −Rf junk
   ```

   removes the directory **junk** and all its contents, without prompting.

RATIONALE

For absolute clarity, paragraphs (2b) and (3) in the DESCRIPTION of *rm* describing the behavior when prompting for confirmation, should be interpreted in the following manner:

```
if ((NOT f_option) AND
    ((not_writable AND input_is_terminal) OR i_option))
```

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application not using the −**f** option, or using the −**i** option, relies on the system to provide the most suitable dialog directly with the user, based on the behavior specified.

The −**r** option is historical practice on all known systems. The synonym −**R** option is provided for consistency with the other utilities in this volume of IEEE Std 1003.1-200x that provide options requesting recursive descent through the file hierarchy.

The behavior of the −**f** option in historical versions of *rm* is inconsistent. In general, along with ''forcing'' the unlink without prompting for permission, it always causes diagnostic messages to be suppressed and the exit status to be unmodified for nonexistent operands and files that cannot be unlinked. In some versions, however, the −**f** option suppresses usage messages and system errors as well. Suppressing such messages is not a service to either shell scripts or users.

It is less clear that error messages regarding files that cannot be unlinked (removed) should be suppressed. Although this is historical practice, this volume of IEEE Std 1003.1-200x does not permit the −**f** option to suppress such messages.

When given the −**r** and −**i** options, historical versions of *rm* prompt the user twice for each directory, once before removing its contents and once before actually attempting to delete the directory entry that names it. This allows the user to ''prune'' the file hierarchy walk. Historical versions of *rm* were inconsistent in that some did not do the former prompt for directories named on the command line and others had obscure prompting behavior when the −**i** option was specified and the permissions of the file did not permit writing. The POSIX Shell and Utilities *rm* differs little from historic practice, but does require that prompts be consistent. Historical versions of *rm* were also inconsistent in that prompts were done to both standard output and standard error. This volume of IEEE Std 1003.1-200x requires that prompts be done to standard error, for consistency with *cp* and *mv*, and to allow historical extensions to *rm* that provide an option to list deleted files on standard output.

The *rm* utility is required to descend to arbitrary depths so that any file hierarchy may be deleted. This means, for example, that the *rm* utility cannot run out of file descriptors during its descent (that is, if the number of file descriptors is limited, *rm* cannot be implemented in the historical fashion where one file descriptor is used per directory level). Also, *rm* is not permitted to fail because of path length restrictions, unless an operand specified by the user is longer than {PATH_MAX}.

The *rm* utility removes symbolic links themselves, not the files they refer to, as a consequence of the dependence on the *unlink*( ) functionality, per the DESCRIPTION. When removing hierarchies with −**r** or −**R**, the prohibition on following symbolic links has to be made explicit.

FUTURE DIRECTIONS

None.

SEE ALSO

rmdir, the System Interfaces volume of IEEE Std 1003.1-200x, *remove*( ), *unlink*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

Text is added to clarify actions relating to symbolic links as specified in the IEEE P1003.2b draft standard.

NAME

rmdel — remove a delta from an SCCS file (**DEVELOPMENT**)

SYNOPSIS

XSI
```
rmdel −r SID file...
```

DESCRIPTION

The *rmdel* utility shall remove the delta specified by the SID from each named SCCS file. The delta to be removed shall be the most recent delta in its branch in the delta chain of each named SCCS file. In addition, the application shall ensure that the SID specified is not that of a version being edited for the purpose of making a delta; that is, if a *p-file* (see *get* (on page 2675)) exists for the named SCCS file, the SID specified shall not appear in any entry of the *p-file*.

Removal of a delta shall be restricted to:

1. The user who made the delta
2. The owner of the SCCS file
3. The owner of the directory containing the SCCS file

OPTIONS

The *rmdel* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−r *SID* Specify the SCCS identification string (*SID*) of the delta to be deleted.

OPERANDS

The following operand shall be supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, the *rmdel* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files shall be silently ignored.

If exactly one *file* operand appears, and it is `'−'`, the standard input shall be read; each line of the standard input is taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only when the *file* operand is specified as `'−'`. Each line of the text file shall be interpreted as an SCCS pathname.

INPUT FILES

The SCCS files shall be files of unspecified format. |

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *rmdel*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The SCCS files shall be files of unspecified format. During processing of a *file*, a temporary *x-file*, as described in *admin* (on page 2328), may be created and deleted; a locking *z-file*, as described in *get* (on page 2675), may be created and deleted.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delta, *get*, *prs*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

NAME

rmdir — remove directories

SYNOPSIS

`rmdir [−p] dir...`

DESCRIPTION

The *rmdir* utility shall remove the directory entry specified by each *dir* operand, which, in order to succeed, the application shall ensure refers to an empty directory.

Directories shall be processed in the order specified. If a directory and a subdirectory of that directory are specified in a single invocation of the *rmdir* utility, the application shall specify the subdirectory before the parent directory so that the parent directory will be empty when the *rmdir* utility tries to remove it.

OPTIONS

The *rmdir* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−p Remove all directories in a pathname. For each *dir* operand:

1. The directory entry it names shall be removed.
2. If the *dir* operand includes more than one pathname component, effects equivalent to the following command shall occur:

```
rmdir −p $(dirname dir)
```

OPERANDS

The following operand shall be supported:

dir A pathname of an empty directory to be removed.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *rmdir*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Each directory entry specified by a *dir* operand was removed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The definition of an empty directory is one that contains, at most, directory entries for dot and dot-dot.

EXAMPLES

If a directory **a** in the current directory is empty except it contains a directory **b** and **a/b** is empty except it contains a directory **c**:

```
rmdir −p a/b/c
```

removes all three directories.

RATIONALE

On historical System V systems, the −**p** option also caused a message to be written to the standard output. The message indicated whether the whole path was removed or whether part of the path remained for some reason. The STDERR section requires this diagnostic when the entire path specified by a *dir* operand is not removed, but does not allow the status message reporting success to be written as a diagnostic.

The *rmdir* utility on System V also included an −**s** option that suppressed the informational message output by the −**p** option. This option has been omitted because the informational message is not specified by this volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

rm, the System Interfaces volume of IEEE Std 1003.1-200x, *remove*( ), *rmdir*( ), *unlink*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

sact — print current SCCS file-editing activity (**DEVELOPMENT**)

SYNOPSIS

XSI

```
sact file...
```

DESCRIPTION

The *sact* utility shall inform the user of any impending deltas to a named SCCS file by writing a list to standard output. This situation occurs when *get* −**e** has been executed previously without a subsequent execution of *delta*, *unget*, or *sccs* **unedit**.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

file	A pathname of an existing SCCS file or a directory. If *file* is a directory, the *sact* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files shall be silently ignored. If exactly one *file* operand appears, and it is `'−'`, the standard input shall be read; each line of the standard input shall be taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only when the *file* operand is specified as `'−'`. Each line of the text file shall be interpreted as an SCCS pathname.

INPUT FILES

Any SCCS files interrogated are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *sact*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output for each named file shall consist of a line in the following format:

`"%sΔ%sΔ%sΔ%sΔ%s\n", <SID>, <new SID>, <login>, <date>, <time>`

<SID>	Specifies the SID of a delta that currently exists in the SCCS file to which changes are made to make the new delta.
<new SID>	Specifies the SID for the new delta to be created.
<login>	Contains the login name of the user who makes the delta (that is, who executed a *get* for editing).
<date>	Contains the date that *get* −**e** was executed, in the format used by the *prs* **:D:** data keyword.
<time>	Contains the time that *get* −**e** was executed, in the format used by the *prs* **:T:** data keyword.

If there is more than one named file or if a directory or standard input is named, each pathname shall be written before each of the preceding lines:

`"\n%s:\n", <pathname>`

STDERR

The standard error shall be used only for optional informative messages concerning SCCS files with no impending deltas, and for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delta, *get*, *unget*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

NAME

sccs — front end for the SCCS subsystem (**DEVELOPMENT**)

SYNOPSIS

XSI

```
sccs [−r][−d path][−p path] command [options...][operands...]
```

DESCRIPTION

The *sccs* utility is a front end to the SCCS programs. It also includes the capability to run set-user-id to another user to provide additional protection.

The *sccs* utility shall invoke the specified *command* with the specified *options* and *operands*. By default, each of the *operands* shall be modified by prefixing it with the string `"SCCS/s."`.

The *command* can be the name of one of the SCCS utilities in this volume of IEEE Std 1003.1-200x (*admin*, *delta*, *get*, *prs*, *rmdel*, *sact*, *unget*, *val*, or *what*) or one of the pseudo-utilities listed in the EXTENDED DESCRIPTION section.

OPTIONS

The *sccs* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that *options* operands are actually options to be passed to the utility named by *command*. When the portion of the command:

```
command [options ... ] [operands ... ]
```

is considered, all of the pseudo-utilities used as *command* shall support the Utility Syntax Guidelines. Any of the other SCCS utilities that can be invoked in this manner support the Guidelines to the extent indicated by their individual OPTIONS sections.

The following options shall be supported preceding the *command* operand:

−d *path* A pathname of a directory to be used as a root directory for the SCCS files. The default shall be the current directory. The **−d** option shall take precedence over the *PROJECTDIR* variable. See **−p**.

−p *path* A pathname of a directory in which the SCCS files are located. The default shall be the **SCCS** directory.

The **−p** option differs from the **−d** option in that the **−d** option-argument shall be prefixed to the entire pathname and the **−p** option-argument shall be inserted before the final component of the pathname. For example:

```
sccs −d /x −p y get a/b
```

converts to:

```
get /x/a/y/s.b
```

This allows the creation of aliases such as:

```
alias syssccs="sccs −d /usr/src"
```

which is used as:

```
syssccs get cmd/who.c
```

−r Invoke *command* with the real user ID of the process, not any effective user ID that the *sccs* utility is set to. Certain commands (*admin*, **check**, **clean**, **diffs**, **info**, *rmdel*, and **tell**) cannot be run set-user-ID by all users, since this would allow anyone to change the authorizations. These commands are always run as the real user.

OPERANDS

The following operands shall be supported:

command	An SCCS utility name or the name of one of the pseudo-utilities listed in the EXTENDED DESCRIPTION section.
options	An option or option-argument to be passed to *command*.
operands	An operand to be passed to *command*.

STDIN

See the utility description for the specified *command*.

INPUT FILES

See the utility description for the specified *command*.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *sccs*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PROJECTDIR	Provide a default value for the **−d** *path* option. If the value of *PROJECTDIR* begins with a slash, it shall be considered an absolute pathname; otherwise, the value of *PROJECTDIR* is treated as a user name and that user's initial working directory shall be examined for a subdirectory **src** or **source**. If such a directory is found, it shall be used. Otherwise, the value shall be used as a relative pathname.

Additional environment variable effects may be found in the utility description for the specified *command*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the utility description for the specified *command*.

STDERR

See the utility description for the specified *command*.

OUTPUT FILES

See the utility description for the specified *command*.

EXTENDED DESCRIPTION

The following pseudo-utilities shall be supported as *command* operands. All options referred to in the following list are values given in the *options* operands following *command*.

check	Equivalent to **info**, except that nothing shall be printed if nothing is being edited, and a non-zero exit status shall be returned if anything is being edited. The intent is to have this included in an ‘‘install’’ entry in a makefile to ensure that everything is included into the SCCS file before a version is installed.
clean	Remove everything from the current directory that can be recreated from SCCS files, but do not remove any files being edited. If the −**b** option is given, branches shall be ignored in the determination of whether they are being edited; this is dangerous if branches are kept in the same directory.
create	Create an SCCS file, taking the initial contents from the file of the same name. Any options to *admin* are accepted. If the creation is successful, the original files shall be renamed by prefixing the basenames with a comma. These renamed files should be removed after it has been verified that the SCCS files have been created successfully.
delget	Perform a *delta* on the named files and then *get* new versions. The new versions shall have ID keywords expanded and shall not be editable. Any −**m**, −**p**, −**r**, −**s**, and −**y** options shall be passed to *delta*, and any −**b**, −**c**, −**e**, −**i**, −**k**, −**l**, −**s**, and −**x** options shall be passed to *get*.
deledit	Equivalent to **delget**, except that the *get* phase shall include the −**e** option. This option is useful for making a checkpoint of the current editing phase. The same options shall be passed to *delta* as described above, and all the options listed for *get* above except −**e** shall be passed to **edit**.
diffs	Write a difference listing between the current version of the files checked out for editing and the versions in SCCS format. Any −**r**, −**c**, −**i**, −**x**, and −**t** options shall be passed to *get*; any −**l**, −**s**, −**e**, −**f**, −**h**, and −**b** options shall be passed to *diff*. A −**C** option shall be passed to *diff* as −**c**.
edit	Equivalent to *get* −**e**.
fix	Remove the named delta, but leave a copy of the delta with the changes that were in it. It is useful for fixing small compiler bugs, and so on. The application shall ensure that it is followed by a −**r** *SID* option. Since **fix** does not leave audit trails, it should be used carefully.
info	Write a listing of all files being edited. If the −**b** option is given, branches (that is, SIDs with two or fewer components) shall be ignored. If a −**u** *user* option is given, then only files being edited by the named user shall be listed. A −**U** option shall be equivalent to −**u**<*current user*>.
print	Write out verbose information about the named files, equivalent to *sccs prs*.
tell	Write a <newline>-separated list of the files being edited to standard output. Takes the −**b**, −**u**, and −**U** options like **info** and **check**.
unedit	This is the opposite of an **edit** or a *get* −**e**. It should be used with caution, since any changes made since the *get* are lost.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Many of the SCCS utilities take directory names as operands as well as specific filenames. The pseudo-utilities supported by *sccs* are not described as having this capability, but are not prohibited from doing so.

EXAMPLES

1. To get a file for editing, edit it and produce a new delta:

```
sccs get −e file.c
ex file.c
sccs delta file.c
```

2. To get a file from another directory:

```
sccs −p /usr/src/sccs/s. get cc.c
```

or:

```
sccs get /usr/src/sccs/s.cc.c
```

3. To make a delta of a large number of files in the current directory:

```
sccs delta *.c
```

4. To get a list of files being edited that are not on branches:

```
sccs info −b
```

5. To delta everything being edited by the current user:

```
sccs delta $(sccs tell −U)
```

6. In a makefile, to get source files from an SCCS file if it does not already exist:

```
SRCS = <list of source files>
$(SRCS):
    sccs get $(REL) $@
```

RATIONALE

SCCS and its associated utilities are part of the XSI Development Utilities option within the XSI extension.

SCCS is an abbreviation for Source Code Control System. It is a maintenance and enhancement tracking tool. When a file is put under SCCS, the source code control system maintains the file and, when changes are made, identifies and stores them in the file with the original source code and/or documentation. As other changes are made, they too are identified and retained in the file.

Retrieval of the original and any set of changes is possible. Any version of the file as it develops can be reconstructed for inspection or additional modification. History data can be stored with each version, documenting why the changes were made, who made them, and when they were made.

FUTURE DIRECTIONS

None.

SEE ALSO

admin, *delta*, *get*, *make*, *prs*, *rmdel*, *sact*, *unget*, *val*, *what*

CHANGE HISTORY

First released in Issue 4.

Issue 6

In the ENVIRONMENT VARIABLES section, the *PROJECTDIR* description is updated from ‘‘otherwise, the home directory of a user of that name is examined’’ to ‘‘otherwise, the value of *PROJECTDIR* is treated as a user name and that user’s initial working directory is examined’’.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

NAME

sed — stream editor

SYNOPSIS

```
sed [−n] script[file...]
sed [−n][−e script]...[−f script_file]...[file...]
```

DESCRIPTION

The *sed* utility is a stream editor that shall read one or more text files, make editing changes according to a script of editing commands, and write the results to standard output. The script shall be obtained from either the *script* operand string or a combination of the option-arguments from the −**e** *script* and −**f** *script_file* options.

OPTIONS

The *sed* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the order of presentation of the −**e** and −**f** options is significant.

The following options shall be supported:

−**e** *script*	Add the editing commands specified by the *script* option-argument to the end of the script of editing commands. The *script* option-argument shall have the same properties as the *script* operand, described in the OPERANDS section.
−**f** *script_file*	Add the editing commands in the file *script_file* to the end of the script.
−**n**	Suppress the default output (in which each line, after it is examined for editing, is written to standard output). Only lines explicitly selected for output are written.

Multiple −**e** and −**f** options may be specified. All commands shall be added to the script in the order specified, regardless of their origin.

OPERANDS

The following operands shall be supported:

file	A pathname of a file whose contents are read and edited. If multiple *file* operands are specified, the named files shall be read in the order specified and the concatenation shall be edited. If no *file* operands are specified, the standard input shall be used.
script	A string to be used as the script of editing commands. The application shall not present a *script* that violates the restrictions of a text file except that the final character need not be a <newline>.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

The input files shall be text files. The *script_file*s named by the −**f** option shall consist of editing commands.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *sed*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements within regular expressions.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), and the behavior of character classes within regular expressions.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The input files shall be written to standard output, with the editing commands specified in the script applied. If the −**n** option is specified, only those input lines selected by the script shall be written to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output files shall be text files whose formats are dependent on the editing commands given.

EXTENDED DESCRIPTION

The *script* shall consist of editing commands of the following form:

```
[address[,address]]function
```

where *function* represents a single-character command verb from the list in **Editing Commands in sed** (on page 3041), followed by any applicable arguments.

The command can be preceded by <blank>s and/or semicolons. The function can be preceded by <blank>s. These optional characters shall have no effect.

In default operation, *sed* cyclically shall append a line of input, less its terminating <newline>, into the pattern space. Normally the pattern space will be empty, unless a **D** command terminated the last cycle. The *sed* utility shall then apply in sequence all commands whose addresses select that pattern space, and at the end of the script copy the pattern space to standard output (except when −**n** is specified) and delete the pattern space. Whenever the pattern space is written to standard output or a named file, *sed* shall immediately follow it with a <newline>.

Some of the editing commands use a hold space to save all or part of the pattern space for subsequent retrieval. The pattern and hold spaces shall each be able to hold at least 8 192 bytes.

Addresses in sed

An address is either a decimal number that counts input lines cumulatively across files, a `'$'` character that addresses the last line of input, or a context address (which consists of a BRE, as described in **Regular Expressions in sed**, preceded and followed by a delimiter, usually a slash).

An editing command with no addresses shall select every pattern space.

An editing command with one address shall select each pattern space that matches the address.

An editing command with two addresses shall select the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line shall be selected.) Starting at the first line following the selected range, *sed* shall look again for the first address. Thereafter, the process shall be repeated. Omitting either or both of the address components in the following form produces undefined results:

`[`*`address`*`[,`*`address`*`]]`

Regular Expressions in sed

The *sed* utility shall support the BREs described in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3, Basic Regular Expressions, with the following additions:

- In a context address, the construction `"\cBREc"`, where *c* is any character other than backslash or <newline>, shall be identical to `"/BRE/"`. If the character designated by *c* appears following a backslash, then it shall be considered to be that literal character, which shall not terminate the BRE. For example, in the context address `"\xabc\xdefx"`, the second *x* stands for itself, so that the BRE is `"abcxdef"`.
- The escape sequence `'\n'` shall match a <newline> embedded in the pattern space. A literal <newline> shall not be used in the BRE of a context address or in the substitute function.
- If an RE is empty (that is, no pattern is specified) *sed* shall behave as if the last RE used in the last command applied (either as an address or as part of a substitute command) was specified.

Editing Commands in sed

In the following list of editing commands, the maximum number of permissible addresses for each function is indicated by [*0addr*], [*1addr*], or [*2addr*], representing zero, one, or two addresses.

The argument *text* shall consist of one or more lines. Each embedded <newline> in the text shall be preceded by a backslash. Other backslashes in text shall be removed, and the following character shall be treated literally.

The **r** and **w** command verbs, and the *w* flag to the **s** command, take an optional *rfile* (or *wfile*) parameter, separated from the command verb letter or flag by one or more <blank>s; implementations may allow zero separation as an extension.

The argument *rfile* or the argument *wfile* shall terminate the editing command. Each *wfile* shall be created before processing begins. Implementations shall support at least ten *wfile* arguments in the script; the actual number (greater than or equal to 10) that is supported by the implementation is unspecified. The use of the *wfile* parameter shall cause that file to be initially created, if it does not exist, or shall replace the contents of an existing file.

The **b**, **r**, **s**, **t**, **w**, **y**, and **:** command verbs shall accept additional arguments. The following synopses indicate which arguments shall be separated from the command verbs by a single

<space>.

The **a** and **r** commands schedule text for later output. The text specified for the **a** command, and the contents of the file specified for the **r** command, shall be written to standard output just before the next attempt to fetch a line of input when executing the **N** or **n** commands, or when reaching the end of the script. If written when reaching the end of the script, and the −**n** option was not specified, the text shall be written after copying the pattern space to standard output. The contents of the file specified for the **r** command shall be as of the time the output is written, not the time the **r** command is applied. The text shall be output in the order in which the **a** and **r** commands were applied to the input.

Command verbs other than **{**, **a**, **b**, **c**, **i**, **r**, **t**, **w**, **:**, and **#** can be followed by a semicolon, optional <blank>s, and another command verb. However, when the **s** command verb is used with the *w* flag, following it with another command in this manner produces undefined results.

A function can be preceded by one or more ′!′ characters, in which case the function shall be applied if the addresses do not select the pattern space. Zero or more <blank>s shall be accepted before the first ′!′ character. It is unspecified whether <blank>s can follow a ′!′ character, and conforming applications shall not follow a ′!′ character with <blank>s.

[_2addr_**] {**_function_ _function_ . . . **}**	Execute a list of *sed* functions only when the pattern space is selected. The list of *sed* functions shall be surrounded by braces and separated by <newline>s, and conform to the following rules. The braces can be preceded or followed by <blank>s. The functions can be preceded by <blank>s, but shall not be followed by <blank>s. The <right-brace> shall be preceded by a <newline> and can be preceded or followed by <blank>s.
[_1addr_**]a** _text_	Write text to standard output as described previously.
[_2addr_**]b [**_label_**]**	Branch to the **:** function bearing the *label*. If *label* is not specified, branch to the end of the script. The implementation shall support *label*s recognized as unique up to at least 8 characters; the actual length (greater than or equal to 8) that shall be supported by the implementation is unspecified. It is unspecified whether exceeding a label length causes an error or a silent truncation.
[_2addr_**]c** _text_	Delete the pattern space. With a 0 or 1 address or at the end of a 2-address range, place *text* on the output and start the next cycle.
[_2addr_**]d**	Delete the pattern space and start the next cycle.
[_2addr_**]D**	Delete the initial segment of the pattern space through the first <newline> and start the next cycle.
[_2addr_**]g**	Replace the contents of the pattern space by the contents of the hold space.
[_2addr_**]G**	Append to the pattern space a <newline> followed by the contents of the hold space.
[_2addr_**]h**	Replace the contents of the hold space with the contents of the pattern space.
[_2addr_**]H**	Append to the hold space a <newline> followed by the contents of the pattern space.

[_1addr_**]i**\
text Write *text* to standard output.

[_2addr_**]l** (The letter ell.) Write the pattern space to standard output in a visually unambiguous form. The characters listed in the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\r'`, `'\t'`, `'\v'`) shall be written as the corresponding escape sequence; the `'\n'` in that table is not applicable. Non-printable characters not in that table shall be written as one three-digit octal number (with a preceding backslash) for each byte in the character (most significant byte first). If the size of a byte on the system is greater than 9 bits, the format used for non-printable characters is implementation-defined.

Long lines shall be folded, with the point of folding indicated by writing a backslash followed by a <newline>; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line shall be marked with a `'$'`.

[_2addr_**]n** Write the pattern space to standard output if the default output has not been suppressed, and replace the pattern space with the next line of input, less its terminating <newline>.

If no next line of input is available, the **n** command verb shall branch to the end of the script and quit without starting a new cycle.

[_2addr_**]N** Append the next line of input, less its terminating <newline>, to the pattern space, using an embedded <newline> to separate the appended material from the original material. Note that the current line number changes.

If no next line of input is available, the **N** command verb shall branch to the end of the script and quit without starting a new cycle or copying the pattern space to standard output.

[_2addr_**]p** Write the pattern space to standard output.

[_2addr_**]P** Write the pattern space, up to the first <newline>, to standard output.

[_1addr_**]q** Branch to the end of the script and quit without starting a new cycle.

[_1addr_**]r** *rfile* Copy the contents of *rfile* to standard output as described previously. If *rfile* does not exist or cannot be read, it shall be treated as if it were an empty file, causing no error condition.

[_2addr_**]s**/*BRE*/*replacement*/*flags*

Substitute the replacement string for instances of the BRE in the pattern space. Any character other than backslash or <newline> can be used instead of a slash to delimit the BRE and the replacement. Within the BRE and the replacement, the BRE delimiter itself can be used as a literal character if it is preceded by a backslash.

The replacement string shall be scanned from beginning to end. An ampersand (`'&'`) appearing in the replacement shall be replaced by the string matching the BRE. The special meaning of `'&'` in this context can be suppressed by preceding it by a backslash. The characters `"\n"`, where *n* is a digit, shall be replaced by the text matched by the corresponding backreference expression. The special meaning of `"\n"` where *n* is a digit in this context, can be suppressed by preceding it by a backslash. For each other backslash (`'\'`) encountered, the following character shall lose its special meaning (if any). The meaning of a `'\'` immediately followed by any character other than '&', '\', a digit, or the delimiter character used for this command, is unspecified.

A line can be split by substituting a <newline> into it. The application shall escape the <newline> in the replacement by preceding it by a backslash. A substitution shall be considered to have been performed even if the replacement string is identical to the string that it replaces. Any backslash used to alter the default meaning of a subsequent character shall be discarded from the BRE or the replacement before evaluating the BRE or using the replacement.

The value of *flags* shall be zero or more of:

n Substitute for the *n*th occurrence only of the BRE found within the pattern space.

g Globally substitute for all non-overlapping instances of the BRE rather than just the first one. If both **g** and **n** are specified, the results are unspecified.

p Write the pattern space to standard output if a replacement was made.

w *wfile* Write. Append the pattern space to *wfile* if a replacement was made. A conforming application shall precede the *wfile* argument with one or more <blank>s. If the *w* flag is not the last flag value given in a concatenation of multiple flag values, the results are undefined.

[*2addr***]t [***label***]**

Test. Branch to the **:** command verb bearing the *label* if any substitutions have been made since the most recent reading of an input line or execution of a **t**. If *label* is not specified, branch to the end of the script.

[*2addr***]w** *wfile*

Append (write) the pattern space to *wfile*.

[*2addr***]x** Exchange the contents of the pattern and hold spaces.

[*2addr***]y/***string1***/***string2***/**

Replace all occurrences of characters in *string1* with the corresponding characters in *string2*. If a backslash followed by an 'n' appear in *string1* or *string2*, the two characters shall be handled as a single <newline>. If the number of characters in *string1* and *string2* are not equal, or if any of the characters in *string1* appear more than once, the results are undefined. Any character other than backslash or <newline> can be used instead of slash to delimit the strings. If the delimiter is not *n*, within *string1* and *string2*, the delimiter itself can be used as a literal character if it is preceded by a backslash. If a backslash character is immediately followed by a backslash character in *string1* or *string2*, the two backslash characters shall be counted as a single literal backslash character. The meaning of a backslash followed by any character that is not 'n', a backslash, or the delimiter character is undefined.

[*0addr***]:***label* Do nothing. This command bears a *label* to which the **b** and **t** commands branch.

[*1addr***]=** Write the following to standard output:

```
"%d\n", <current line number>
```

[*0addr***]** Ignore this empty command.

[*0addr*]# Ignore the `'#'` and the remainder of the line (treat them as a comment), with the single exception that if the first two characters in the script are `"#n"`, the default output shall be suppressed; this shall be the equivalent of specifying −**n** on the command line.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Regular expressions match entire strings, not just individual lines, but a <newline> is matched by `'\n'` in a *sed* RE; a <newline> is not allowed by the general definition of regular expression in IEEE Std 1003.1-200x. Also note that `'\n'` cannot be used to match a <newline> at the end of an arbitrary input line; <newline>s appear in the pattern space as a result of the **N** editing command.

EXAMPLES

This *sed* script simulates the BSD *cat* −**s** command, squeezing excess blank lines from standard input.

```
sed −n '
# Write non-empty lines.
/./ {
    p
    d
    }
# Write a single empty line, then look for more empty lines.
/ˆ$/    p
# Get next line, discard the held <newline> (empty line),
# and look for more empty lines.
:Empty
/ˆ$/    {
    N
    s/.//
    b Empty
    }
# Write the non-empty line before going back to search
# for the first in a set of empty lines.
    p
'
```

RATIONALE

This volume of IEEE Std 1003.1-200x requires implementations to support at least ten distinct *wfile*s, matching historical practice on many implementations. Implementations are encouraged to support more, but conforming applications should not exceed this limit.

The exit status codes specified here are different from those in System V. System V returns 2 for garbled *sed* commands, but returns zero with its usage message or if the input file could not be opened. The standard developers considered this to be a bug.

The manner in which the **l** command writes non-printable characters was changed to avoid the historical backspace-overstrike method, and other requirements to achieve unambiguous output were added. See the RATIONALE for *ed* (on page 2537) for details of the format chosen, which is the same as that chosen for *sed*.

This volume of IEEE Std 1003.1-200x requires implementations to provide pattern and hold spaces of at least 8 192 bytes, larger than the 4 000 bytes spaces used by some historical implementations, but less than the 20 480 bytes limit used in an early proposal. Implementations are encouraged to allocate dynamically larger pattern and hold spaces as needed.

The requirements for acceptance of <blank>s and <space>s in command lines has been made more explicit than in early proposals to describe clearly the historical practice and to remove confusion about the phrase ‘‘protect initial blanks [*sic*] and tabs from the stripping that is done on every script line’’ that appears in much of the historical documentation of the *sed* utility description of text. (Not all implementations are known to have stripped <blank>s from text lines, although they all have allowed leading <blank>s preceding the address on a command line.)

The treatment of `'#'` comments differs from the SVID which only allows a comment as the first line of the script, but matches BSD-derived implementations. The comment character is treated as a command, and it has the same properties in terms of being accepted with leading <blank>s; the BSD implementation has historically supported this.

Early proposals required that a *script_file* have at least one non-comment line. Some historical implementations have behaved in unexpected ways if this were not the case. The standard developers considered that this was incorrect behavior and that application developers should not have to avoid this feature. A correct implementation of this volume of IEEE Std 1003.1-200x shall permit *script_file*s that consist only of comment lines.

Early proposals indicated that if −**e** and −**f** options were intermixed, all −**e** options were processed before any −**f** options. This has been changed to process them in the order presented because it matches historical practice and is more intuitive.

The treatment of the **p** flag to the **s** command differs between System V and BSD-based systems when the default output is suppressed. In the two examples:

```
echo a | sed    's/a/A/p'
echo a | sed −n 's/a/A/p'
```

This volume of IEEE Std 1003.1-200x, BSD, System V documentation, and the SVID indicate that the first example should write two lines with **A**, whereas the second should write one. Some System V systems write the **A** only once in both examples because the **p** flag is ignored if the −**n** option is not specified.

This is a case of a diametrical difference between systems that could not be reconciled through the compromise of declaring the behavior to be unspecified. The SVID/BSD/System V documentation behavior was adopted for this volume of IEEE Std 1003.1-200x because:

- No known documentation for any historic system describes the interaction between the **p** flag and the −**n** option.
- The selected behavior is more correct as there is no technical justification for any interaction between the **p** flag and the −**n** option. A relationship between −**n** and the **p** flag might imply that they are only used together, but this ignores valid scripts that interrupt the cyclical nature of the processing through the use of the **D**, **d**, **q**, or branching commands. Such scripts rely on the **p** suffix to write the pattern space because they do not make use of the default output at the ‘‘bottom’’ of the script.

- Because the −**n** option makes the **p** flag unnecessary, any interaction would only be useful if *sed* scripts were written to run both with and without the −**n** option. This is believed to be unlikely. It is even more unlikely that programmers have coded the **p** flag expecting it to be unnecessary. Because the interaction was not documented, the likelihood of a programmer discovering the interaction and depending on it is further decreased.

- Finally, scripts that break under the specified behavior produce too much output instead of too little, which is easier to diagnose and correct.

The form of the substitute command that uses the **n** suffix was limited to the first 512 matches in an early proposal. This limit has been removed because there is no reason an editor processing lines of {LINE_MAX} length should have this restriction. The command **s/a/A/2047** should be able to substitute the 2 047th occurrence of **a** on a line.

The **b**, **t**, and **:** commands are documented to ignore leading white space, but no mention is made of trailing white space. Historical implementations of *sed* assigned different locations to the labels `'x'` and `"x "`. This is not useful, and leads to subtle programming errors, but it is historical practice, and changing it could theoretically break working scripts. Implementors are encouraged to provide warning messages about labels that are never used or jumps to labels that do not exist.

Historically, the *sed* **!** and **}** editing commands did not permit multiple commands on a single line using a semicolon as a command delimiter. Implementations are permitted, but not required, to support this extension.

FUTURE DIRECTIONS

None.

SEE ALSO

awk, *ed*, *grep*

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- Implementations are required to support at least ten *wfile* arguments in an editing command.

The EXTENDED DESCRIPTION is changed to align with the IEEE P1003.2b draft standard.

IEEE PASC Interpretation 1003.2 #190 is applied.

IEEE PASC Interpretation 1003.2 #203 is applied, clarifying the meaning of the backslash escape sequences in a replacement string for a BRE.

NAME

sh — shell, the standard command language interpreter

SYNOPSIS

```
sh [−abCefhimnuvx][−o option][+abCefhmnuvx][+o option]
    [command_file [argument...]]

sh −c[−abCefhimnuvx][−o option][+abCefhimnuvx][+o option]command_string
    [command_name [argument...]]

sh −s[−abCefhimnuvx][−o option][+abCefhimnuvx][+o option][argument]
```

DESCRIPTION

The *sh* utility is a command language interpreter that shall execute commands read from a command line string, the standard input, or a specified file. The application shall ensure that the commands to be executed are expressed in the language described in Chapter 2 (on page 2231).

Pathname expansion shall not fail due to the size of a file.

Shell input and output redirections have an implementation-defined offset maximum that is established in the open file description.

OPTIONS

The *sh* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, with an extension for support of a leading plus sign (`'+'`) as noted below.

The **−a**, **−b**, **−C**, **−e**, **−f**, **−m**, **−n**, **−o** *option*, **−u**, **−v**, and **−x** options are described as part of the *set* utility in Section 2.14 (on page 2266). The option letters derived from the *set* special built-in shall also be accepted with a leading plus sign (`'+'`) instead of a leading hyphen (meaning the reverse case of the option as described in this volume of IEEE Std 1003.1-200x).

The following additional options shall be supported:

−c	Read commands from the *command_string* operand. Set the value of special parameter 0 (see Section 2.5.2 (on page 2235)) from the value of the *command_name* operand and the positional parameters ($1, $2, and so on) in sequence from the remaining *argument* operands. No commands shall be read from the standard input.
−i	Specify that the shell is *interactive*; see below. An implementation may treat specifying the **−i** option as an error if the real user ID of the calling process does not equal the effective user ID or if the real group ID does not equal the effective group ID.
−s	Read commands from the standard input.

If there are no operands and the **−c** option is not specified, the **−s** option shall be assumed.

If the **−i** option is present, or if there are no operands and the shell's standard input and standard error are attached to a terminal, the shell is considered to be *interactive*.

OPERANDS

The following operands shall be supported:

−	A single hyphen shall be treated as the first operand and then ignored. If both `'−'` and `"−−"` are given as arguments, or if other operands precede the single hyphen, the results are undefined.
argument	The positional parameters ($1, $2, and so on) shall be set to *arguments*, if any.

command_file The pathname of a file containing commands. If the pathname contains one or more slash characters, the implementation attempts to read that file; the file need not be executable. If the pathname does not contain a slash character:

- The implementation shall attempt to read that file from the current working directory; the file need not be executable.
- If the file is not in the current working directory, the implementation may perform a search for an executable file using the value of *PATH*, as described in Section 2.9.1.1 (on page 2249).

Special parameter 0 (see Section 2.5.2 (on page 2235)) shall be set to the value of *command_file*. If *sh* is called using a synopsis form that omits *command_file*, special parameter 0 shall be set to the value of the first argument passed to *sh* from its parent (for example, *argv*[0] for a C program), which is normally a pathname used to execute the *sh* utility.

command_name

A string assigned to special parameter 0 when executing the commands in *command_string*. If *command_name* is not specified, special parameter 0 shall be set to the value of the first argument passed to *sh* from its parent (for example, *argv*[0] for a C program), which is normally a pathname used to execute the *sh* utility.

command_string

A string that shall be interpreted by the shell as one or more commands, as if the string were the argument to the *system*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x. If the *command_string* operand is an empty string, *sh* shall exit with a zero exit status.

STDIN

The standard input shall be used only if one of the following is true:

- The −**s** option is specified.
- The −**c** option is not specified and no operands are specified.
- The script executes one or more commands that require input from standard input (such as a *read* command that does not redirect its input).

See the INPUT FILES section.

When the shell is using standard input and it invokes a command that also uses standard input, the shell shall ensure that the standard input file pointer points directly after the command it has read when the command begins execution. It shall not read ahead in such a manner that any characters intended to be read by the invoked command are consumed by the shell (whether interpreted by the shell or not) or that characters that are not read by the invoked command are not seen by the shell. When the command expecting to read standard input is started asynchronously by an interactive shell, it is unspecified whether characters are read by the command or interpreted by the shell.

If the standard input to *sh* is a FIFO or terminal device and is set to non-blocking reads, then *sh* shall enable blocking reads on standard input. This shall remain in effect when the command completes.

INPUT FILES

The input file shall be a text file, except that line lengths shall be unlimited. If the input file is empty or consists solely of blank lines or comments, or both, *sh* shall exit with a zero exit status.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *sh*:

ENV	This variable, when and only when an interactive shell is invoked, shall be subjected to parameter expansion (see Section 2.6.2 (on page 2239)) by the shell, and the resulting value shall be used as a pathname of a file containing shell commands to execute in the current environment. The file need not be executable. If the expanded value of *ENV* is not an absolute pathname, the results are unspecified. *ENV* shall be ignored if the real and effective user IDs or real and effective group IDs of the process are different.
FCEDIT	This variable, when expanded by the shell, shall determine the default value for the −**e** *editor* option's *editor* option-argument. If *FCEDIT* is null or unset, *ed* shall be used as the editor. This volume of IEEE Std 1003.1-200x specifies the effects of this variable only for systems supporting the User Portability Utilities option.
HISTFILE	Determine a pathname naming a command history file. If the *HISTFILE* variable is not set, the shell may attempt to access or create a file **.sh_history** in the directory referred to by the *HOME* environment variable. If the shell cannot obtain both read and write access to, or create, the history file, it shall use an unspecified mechanism that allows the history to operate properly. (References to history ''file'' in this section shall be understood to mean this unspecified mechanism in such cases.) An implementation may choose to access this variable only when initializing the history file; this initialization shall occur when *fc* or *sh* first attempt to retrieve entries from, or add entries to, the file, as the result of commands issued by the user, the file named by the *ENV* variable, or implementation-defined system start-up files. Implementations may choose to disable the history list mechanism for users with appropriate privileges who do not set *HISTFILE*; the specific circumstances under which this occurs are implementation-defined. If more than one instance of the shell is using the same history file, it is unspecified how updates to the history file from those shells interact. As entries are deleted from the history file, they shall be deleted oldest first. It is unspecified when history file entries are physically removed from the history file. This volume of IEEE Std 1003.1-200x specifies the effects of this variable only for systems supporting the User Portability Utilities option.
HISTSIZE	Determine a decimal number representing the limit to the number of previous commands that are accessible. If this variable is unset, an unspecified default greater than or equal to 128 shall be used. The maximum number of commands in the history list is unspecified, but shall be at least 128. An implementation may choose to access this variable only when initializing the history file, as described under *HISTFILE*. Therefore, it is unspecified whether changes made to *HISTSIZE* after the history file has been initialized are effective.
HOME	Determine the pathname of the user's home directory. The contents of *HOME* are used in Tilde Expansion as described in Section 2.6.1 (on page 2239). This volume of IEEE Std 1003.1-200x specifies the effects of this variable only for systems supporting the User Portability Utilities option.
IFS	*Input field separators*: a string treated as a list of characters that shall be used for field splitting and to split lines into words with the *read* command. See Section 2.6.5 (on page 2243). If *IFS* is not set, the shell shall behave as if the value of *IFS* were <space>, <tab>, and <newline>. Implementations may ignore the value of *IFS* in the environment at the time *sh* is invoked, treating *IFS* as if it were not set.

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_COLLATE	Determine the behavior of range expressions, equivalence classes and multi-character collating elements within pattern matching.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), which characters are defined as letters (character class **alpha**), and the behavior of character classes within pattern matching.
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
	MAIL	Determine a pathname of the user's mailbox file for purposes of incoming mail notification. If this variable is set, the shell shall inform the user if the file named by the variable is created or if its modification time has changed. Informing the user shall be accomplished by writing a string of unspecified format to standard error prior to the writing of the next primary prompt string. Such check shall be performed only after the completion of the interval defined by the *MAILCHECK* variable after the last such check. The user shall be informed only if *MAIL* is set and *MAILPATH* is not set. This volume of IEEE Std 1003.1-200x specifies the effects of this variable only for systems supporting the User Portability Utilities option.
	MAILCHECK	Establish a decimal integer value that specifies how often (in seconds) the shell shall check for the arrival of mail in the files specified by the *MAILPATH* or *MAIL* variables. The default value shall be 600 seconds. If set to zero, the shell shall check before issuing each primary prompt. This volume of IEEE Std 1003.1-200x specifies the effects of this variable only for systems supporting the User Portability Utilities option.
	MAILPATH	Provide a list of pathnames and optional messages separated by colons. If this variable is set, the shell shall inform the user if any of the files named by the variable are created or if any of their modification times change. (See the preceding entry for *MAIL* for descriptions of mail arrival and user informing.) Each pathname can be followed by `'%'` and a string that shall be subjected to parameter expansion and written to standard error when the modification time changes. If a `'%'` character in the pathname is preceded by a backslash, it shall be treated as a literal `'%'` in the pathname. The default message is unspecified. The *MAILPATH* environment variable takes precedence over the *MAIL* variable. This volume of IEEE Std 1003.1-200x specifies the effects of this variable only for systems supporting the User Portability Utilities option.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
	PATH	Establish a string formatted as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables, used to effect command interpretation; see Section 2.9.1.1 (on page 2249).

PWD This variable shall represent an absolute pathname of the current working directory. Assignments to this variable may be ignored unless the value is an absolute pathname of the current working directory and there are no filename components of dot or dot-dot.

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the STDERR section.

STDERR

Except as otherwise stated (by the descriptions of any invoked utilities or in interactive mode), standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

See Chapter 2. The following additional capabilities are supported on systems supporting the User Portability Utilities option.

Command History List

When the *sh* utility is being used interactively, it shall maintain a list of commands previously entered from the terminal in the file named by the *HISTFILE* environment variable. The type, size, and internal format of this file are unspecified. Multiple *sh* processes can share access to the file for a user, if file access permissions allow this; see the description of the *HISTFILE* environment variable.

Command Line Editing

When *sh* is being used interactively from a terminal, the current command and the command history (see *fc* (on page 2637)) can be edited using *vi*-mode command line editing. This mode uses commands, described below, similar to a subset of those described in the *vi* utility. Implementations may offer other command line editing modes corresponding to other editing utilities.

The command *set* −**o** *vi* shall enable *vi*-mode editing and place *sh* into *vi* insert mode (see **Command Line Editing (vi-mode)** (on page 3053)). This command also shall disable any other editing mode that the implementation may provide. The command *set* **+o** *vi* disables *vi*-mode editing.

Certain block-mode terminals may be unable to support shell command line editing. If a terminal is unable to provide either edit mode, it need not be possible to *set* −**o** *vi* when using the shell on this terminal.

In the following sections, the characters *erase*, *interrupt*, *kill*, and *end-of-file* are those set by the *stty* utility.

Command Line Editing (vi-mode)

In *vi* editing mode, there shall be a distinguished line, the edit line. All the editing operations which modify a line affect the edit line. The edit line is always the newest line in the command history buffer.

With *vi*-mode enabled, *sh* can be switched between insert mode and command mode.

When in insert mode, an entered character shall be inserted into the command line, except as noted in **vi Line Editing Insert Mode**. Upon entering *sh* and after termination of the previous command, *sh* shall be in insert mode.

Typing an escape character shall switch *sh* into command mode (see **vi Line Editing Command Mode** (on page 3054)). In command mode, an entered character shall either invoke a defined operation, is used as part of a multi-character operation, or is treated as an error. A character that is not recognized as part of an editing command shall terminate any specific editing command and shall alert the terminal. Typing the *interrupt* character in command mode shall cause *sh* to terminate command line editing on the current command line, reissue the prompt on the next line of the terminal, and reset the command history (see *fc* (on page 2637)) so that the most recently executed command is the previous command (that is, the command that was being edited when it was interrupted is not reentered into the history).

In the following sections, the phrase ‘‘move the cursor to the beginning of the word’’ shall mean ‘‘move the cursor to the first character of the current word’’ and the phrase ‘‘move the cursor to the end of the word’’ shall mean ‘‘move the cursor to the last character of the current word’’. The phrase ‘‘beginning of the command line’’ indicates the point between the end of the prompt string issued by the shell (or the beginning of the terminal line, if there is no prompt string) and the first character of the command text.

vi Line Editing Insert Mode

While in insert mode, any character typed shall be inserted in the current command line, unless it is from the following set.

<newline>	Execute the current command line. If the current command line is not empty, this line shall be entered into the command history (see *fc*).
erase	Delete the character previous to the current cursor position and move the current cursor position back one character. In insert mode, characters shall be erased from both the screen and the buffer when backspacing.
interrupt	Terminate command line editing with the same effects as described for interrupting command mode; see **Command Line Editing (vi-mode)**.
kill	Clear all the characters from the input line.
<control>-V	Insert the next character input, even if the character is otherwise a special insert mode character.
<control>-W	Delete the characters from the one preceding the cursor to the preceding word boundary. The word boundary in this case is the closer to the cursor of either the beginning of the line or a character that is in neither the **blank** nor **punct** character classification of the current locale.
end-of-file	Interpreted as the end of input in *sh*. This interpretation shall occur only at the beginning of an input line. If *end-of-file* is entered other than at the beginning of the line, the results are unspecified.

<ESC> Place *sh* into command mode.

vi Line Editing Command Mode

In command mode for the command line editing feature, decimal digits not beginning with 0 that precede a command letter shall be remembered. Some commands use these decimal digits as a count number that affects the operation.

The term *motion command* represents one of the commands:

`<space>  0  b  F  l  W  ˆ  $  ;  E  f  T  w  |  ,  B  e  h  t`

If the current line is not the edit line, any command that modifies the current line shall cause the content of the current line to replace the content of the edit line, and the current line shall become the edit line. This replacement cannot be undone (see the **u** and **U** commands below). The modification requested shall then be performed to the edit line. When the current line is the edit line, the modification shall be done directly to the edit line.

Any command that is preceded by *count* shall take a count (the numeric value of any preceding decimal digits). Unless otherwise noted, this count shall cause the specified operation to repeat by the number of times specified by the count. Also unless otherwise noted, a *count* that is out of range is considered an error condition and shall alert the terminal, but neither the cursor position, nor the command line, shall change.

The terms *word* and *bigword* are used as defined in the *vi* description. The term *save buffer* corresponds to the term *unnamed buffer* in *vi*.

The following commands shall be recognized in command mode:

<newline> Execute the current command line. If the current command line is not empty, this line shall be entered into the command history (see *fc*).

<control>-L Redraw the current command line. Position the cursor at the same location on the redrawn line.

Insert the character '#' at the beginning of the current command line and treat the resulting edit line as a comment. This line shall be entered into the command history; see *fc* (on page 2637).

= Display the possible shell word expansions (see Section 2.6 (on page 2238)) of the bigword at the current command line position.

> **Note:** This does not modify the content of the current line, and therefore does not cause the current line to become the edit line.

These expansions shall be displayed on subsequent terminal lines. If the bigword contains none of the characters '?', '*', or '[', an asterisk ('*') shall be implicitly assumed at the end. If any directories are matched, these expansions shall have a '/' character appended. After the expansion, the line shall be redrawn, the cursor is repositioned at the current cursor position, and *sh* shall be placed in command mode.

**** Perform pathname expansion (see Section 2.6.6 (on page 2244)) on the current bigword, up to the largest set of characters that can be matched uniquely. If the bigword contains none of the characters '?', '*', or '[', an asterisk ('*') shall be implicitly assumed at the end. This maximal expansion then shall replace the original bigword in the command line, and the cursor shall be placed after this expansion. If the resulting bigword completely and uniquely matches a directory, a '/' character shall be inserted directly after the bigword. If some other file is completely matched, a single <space> shall be inserted after the bigword. After this operation, *sh* shall be placed in insert mode.

*****
Perform pathname expansion on the current bigword and insert all expansions into the command to replace the current bigword, with each expansion separated by a single <space>. If at the end of the line, the current cursor position shall be moved to the first column position following the expansions and *sh* shall be placed in insert mode. Otherwise, the current cursor position shall be the last column position of the first character after the expansions and *sh* shall be placed in insert mode. If the current bigword contains none of the characters '?', '*', or '[', before the operation, an asterisk shall be implicitly assumed at the end.

@*letter*
Insert the value of the alias named _*letter*. The symbol *letter* represents a single alphabetic character from the portable character set; implementations may support additional characters as an extension. If the alias _*letter* contains other editing commands, these commands shall be performed as part of the insertion. If no alias _*letter* is enabled, this command shall have no effect.

[*count*]~
Convert, if the current character is a lowercase letter, to the equivalent uppercase letter and *vice versa*, as prescribed by the current locale. The current cursor position then shall be advanced by one character. If the cursor was positioned on the last character of the line, the case conversion shall occur, but the cursor shall not advance. If the '~' command is preceded by a *count*, that number of characters shall be converted, and the cursor shall be advanced to the character position after the last character converted. If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.

[*count*].
Repeat the most recent non-motion command, even if it was executed on an earlier command line. If the previous command was preceded by a *count*, and no count is given on the '.' command, the count from the previous command shall be included as part of the repeated command. If the '.' command is preceded by a *count*, this shall override any *count* argument to the previous command. The *count* specified in the '.' command shall become the count for subsequent '.' commands issued without a count.

[*number*]v
Invoke the *vi* editor to edit the current command line in a temporary file. When the editor exits, the commands in the temporary file shall be executed and placed in the command history. If a *number* is included, it specifies the command number in the command history to be edited, rather than the current command line.

[*count*]l (ell)
[*count*]<space>
Move the current cursor position to the next character position. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.

[*count*]h
Move the current cursor position to the *count*th (default 1) previous character position. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the count is larger than the number of characters before the cursor, this shall not be considered an error; the cursor shall move to the first character on the line.

[*count*]w
Move to the start of the next word. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be

	advanced. If the *count* is larger than the number of words after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
[*count*]**W**	Move to the start of the next bigword. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of bigwords after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
[*count*]**e**	Move to the end of the current word. If at the end of a word, move to the end of the next word. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of words after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
[*count*]**E**	Move to the end of the current bigword. If at the end of a bigword, move to the end of the next bigword. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the *count* is larger than the number of bigwords after the cursor, this shall not be considered an error; the cursor shall advance to the last character on the line.
[*count*]**b**	Move to the beginning of the current word. If at the beginning of a word, move to the beginning of the previous word. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the *count* is larger than the number of words preceding the cursor, this shall not be considered an error; the cursor shall return to the first character on the line.
[*count*]**B**	Move to the beginning of the current bigword. If at the beginning of a bigword, move to the beginning of the previous bigword. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the *count* is larger than the number of bigwords preceding the cursor, this shall not be considered an error; the cursor shall return to the first character on the line.
^	Move the current cursor position to the first character on the input line that is not a <blank>.
$	Move to the last character position on the current command line.
0	(Zero.) Move to the first character position on the current command line.
[*count*] **|**	Move to the *count*th character position on the current command line. If no number is specified, move to the first position. The first character position shall be numbered 1. If the count is larger than the number of characters on the line, this shall not be considered an error; the cursor shall be placed on the last character on the line.
[*count*]**f***c*	Move to the first occurrence of the character `'c'` that occurs after the current cursor position. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the character `'c'` does not occur in the line after the current cursor position, the terminal shall be alerted and the cursor shall not be moved.
[*count*]**F***c*	Move to the first occurrence of the character `'c'` that occurs before the current cursor position. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the character `'c'`

does not occur in the line before the current cursor position, the terminal shall be alerted and the cursor shall not be moved.

[*count***]t***c* Move to the character before the first occurrence of the character `'c'` that occurs after the current cursor position. If the cursor was positioned on the last character of the line, the terminal shall be alerted and the cursor shall not be advanced. If the character `'c'` does not occur in the line after the current cursor position, the terminal shall be alerted and the cursor shall not be moved.

[*count***]T***c* Move to the character after the first occurrence of the character `'c'` that occurs before the current cursor position. If the cursor was positioned on the first character of the line, the terminal shall be alerted and the cursor shall not be moved. If the character `'c'` does not occur in the line before the current cursor position, the terminal shall be alerted and the cursor shall not be moved.

[*count***];** Repeat the most recent **f**, **F**, **t**, or **T** command. Any number argument on that previous command shall be ignored. Errors are those described for the repeated command.

[*count***],** Repeat the most recent **f**, **F**, **t**, or **T** command. Any number argument on that previous command shall be ignored. However, reverse the direction of that command.

a Enter insert mode after the current cursor position. Characters that are entered shall be inserted before the next character.

A Enter insert mode after the end of the current command line.

i Enter insert mode at the current cursor position. Characters that are entered shall be inserted before the current character. |

I Enter insert mode at the beginning of the current command line.

R Enter insert mode, replacing characters from the command line beginning at the current cursor position.

[*count***]c***motion*

Delete the characters between the current cursor position and the cursor position that would result from the specified *motion* command. Then enter insert mode before the first character following any deleted characters. If *count* is specified, it shall be applied to the motion command. A *count* shall be ignored for the following motion commands:

```
0    ^    $    c
```

If the *motion* command is the character `'c'`, the current command line shall be cleared and insert mode shall be entered. If the *motion* command would move the current cursor position toward the beginning of the command line, the character under the current cursor position shall not be deleted. If the motion command would move the current cursor position toward the end of the command line, the character under the current cursor position shall be deleted. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this shall not be considered an error; all of the remaining characters in the aforementioned range shall be deleted and insert mode shall be entered. If the motion command is invalid, the terminal shall be alerted, the cursor shall not be moved, and no text shall be deleted.

C Delete from the current character to the end of the line and enter insert mode at the new end-of-line.

S Clear the entire edit line and enter insert mode.

[*count***]r***c* Replace the current character with the character `'c'`. With a number *count*, replace the current and the following *count*−1 characters. After this command, the current cursor position shall be on the last character that was changed. If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; all of the remaining characters shall be changed.

[*count***]_** Append a <space> after the current character position and then append the last bigword in the previous input line after the <space>. Then enter insert mode after the last character just appended. With a number *count*, append the *count*th bigword in the previous line.

[*count***]x** Delete the character at the current cursor position and place the deleted characters in the save buffer. If the cursor was positioned on the last character of the line, the character shall be deleted and the cursor position shall be moved to the previous character (the new last character). If the *count* is larger than the number of characters after the cursor, this shall not be considered an error; all the characters from the cursor to the end of the line shall be deleted.

[*count***]X** Delete the character before the current cursor position and place the deleted characters in the save buffer. The character under the current cursor position shall not change. If the cursor was positioned on the first character of the line, the terminal shall be alerted, and the **X** command shall have no effect. If the line contained a single character, the **X** command shall have no effect. If the line contained no characters, the terminal shall be alerted and the cursor shall not be moved. If the *count* is larger than the number of characters before the cursor, this shall not be considered an error; all the characters from before the cursor to the beginning of the line shall be deleted.

[*count***]d***motion*

Delete the characters between the current cursor position and the character position that would result from the *motion* command. A number *count* repeats the *motion* command *count* times. If the *motion* command would move toward the beginning of the command line, the character under the current cursor position shall not be deleted. If the *motion* command is **d**, the entire current command line shall be cleared. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this shall not be considered an error; all of the remaining characters in the aforementioned range shall be deleted. The deleted characters shall be placed in the save buffer.

D Delete all characters from the current cursor position to the end of the line. The deleted characters shall be placed in the save buffer.

[*count***]y***motion*

Yank (that is, copy) the characters from the current cursor position to the position resulting from the *motion* command into the save buffer. A number *count* shall be applied to the *motion* command. If the *motion* command would move toward the beginning of the command line, the character under the current cursor position shall not be included in the set of yanked characters. If the *motion* command is **y**, the entire current command line shall be yanked into the save buffer. The current cursor position shall be unchanged. If the *count* is larger than the number of

	characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this shall not be considered an error; all of the remaining characters in the aforementioned range shall be yanked.
Y	Yank the characters from the current cursor position to the end of the line into the save buffer. The current character position shall be unchanged.
[*count***]p**	Put a copy of the current contents of the save buffer after the current cursor position. The current cursor position shall be advanced to the last character put from the save buffer. A *count* shall indicate how many copies of the save buffer shall be put.
[*count***]P**	Put a copy of the current contents of the save buffer before the current cursor position. The current cursor position shall be moved to the last character put from the save buffer. A *count* shall indicate how many copies of the save buffer shall be put.
u	Undo the last command that change the edit line. This operation shall not undo the copy of any command line to the edit line.
U	Undo all changes made to the edit line. This operation shall not undo the copy of any command line to the edit line.
[*count***]k** **[***count***]–**	Set the current command line to be the *count*th previous command line in the shell command history. If *count* is not specified, it shall default to 1. The cursor shall be positioned on the first character of the new command. If a **k** or – command would retreat past the maximum number of commands in effect for this shell (affected by the *HISTSIZE* environment variable), the terminal shall be alerted, and the command shall have no effect.
[*count***]j** **[***count***]+**	Set the current command line to be the *count*th next command line in the shell command history. If *count* is not specified, it shall default to 1. The cursor shall be positioned on the first character of the new command. If a **j** or **+** command advances past the edit line, the current command line shall be restored to the edit line and terminal shall be alerted.
[*number***]G**	Set the current command line to be the oldest command line stored in the shell command history. With a number *number*, set the current command line to be the command line *number* in the history. If command line *number* does not exist, the terminal shall be alerted and the command line shall not be changed.
/*pattern*<newline>	Move backwards through the command history, searching for the specified pattern, beginning with the previous command line. Patterns use the pattern matching notation described in Section 2.13 (on page 2264), except that the '^' character shall have special meaning when it appears as the first character of *pattern*. In this case, the '^' is discarded and the characters after the '^' shall be matched only at the beginning of a line. Commands in the command history shall be treated as strings, not as filenames. If the pattern is not found, the current command line shall be unchanged and the terminal is alerted. If it is found in a previous line, the current command line shall be set to that line and the cursor shall be set to the first character of the new command line.

If *pattern* is empty, the last non-empty pattern provided to **/** or **?** shall be used. If there is no previous non-empty pattern, the terminal shall be alerted and the current command line shall remain unchanged.

?*pattern*<newline>
Move forwards through the command history, searching for the specified pattern, beginning with the next command line. Patterns use the pattern matching notation described in Section 2.13 (on page 2264), except that the '^' character shall have special meaning when it appears as the first character of *pattern*. In this case, the '^' is discarded and the characters after the '^' shall be matched only at the beginning of a line. Commands in the command history shall be treated as strings, not as filenames. If the pattern is not found, the current command line shall be unchanged and the terminal alerted. If it is found in a following line, the current command line shall be set to that line and the cursor shall be set to the fist character of the new command line.

If *pattern* is empty, the last non-empty pattern provided to **/** or **?** shall be used. If there is no previous non-empty pattern, the terminal shall be alerted and the current command line shall remain unchanged.

n Repeat the most recent **/** or **?** command. If there is no previous **/** or **?**, the terminal shall be alerted and the current command line shall remain unchanged.

N Repeat the most recent **/** or **?** command, reversing the direction of the search. If there is no previous **/** or **?**, the terminal shall be alerted and the current command line shall remain unchanged.

EXIT STATUS

The following exit values shall be returned:

0	The script to be executed consisted solely of zero or more blank lines or comments, or both.
1-125	A non-interactive shell detected a syntax, redirection or variable assignment error.
127	A specified *command_file* could not be found by a non-interactive shell.

Otherwise, the shell shall return the exit status of the last command it invoked or attempted to invoke (see also the *exit* utility in Section 2.14 (on page 2266)).

CONSEQUENCES OF ERRORS

See Section 2.8.1 (on page 2247).

APPLICATION USAGE

Standard input and standard error are the files that determine whether a shell is interactive when **−i** is not specified. For example:

```
sh > file
```

and:

```
sh 2> file
```

create interactive and non-interactive shells, respectively. Although both accept terminal input, the results of error conditions are different, as described in Section 2.8.1 (on page 2247); in the second example a redirection error encountered by a special built-in utility aborts the shell.

A conforming application must protect its first operand, if it starts with a plus sign, by preceding it with the "−−" argument that denotes the end of the options.

Applications should note that the standard *PATH* to the shell cannot be assumed to be either **/bin/sh** or **/usr/bin/sh**, and should be determined by interrogation of the *PATH* returned by *getconf PATH*, ensuring that the returned pathname is an absolute pathname and not a shell built in.

For example, to determine the location of the standard *sh* utility:

```
command −v sh
```

On some implementations this might return:

```
/usr/xpg4/bin/sh
```

Furthermore, on systems that support executable scripts (the "#!" construct), it is recommended that applications using executable scripts install them using *getconf* **−v** to determine the shell pathname and update the "#!" script appropriately as it is being installed (for example, with *sed*). For example:

```
#
# Installation time script to install correct POSIX shell pathname
#
# Get list of paths to check
#
Sifs=$IFS
IFS=:
set $(getconf PATH)
IFS=$Sifs
#
# Check each path for ’sh’
#
for i in $@
do
    if [ −f ${i}/sh ];
    then
        Pshell=${i}/sh
    fi
done
#
# This is the list of scripts to update. They should be of the
# form ’${name}.source’ and will be transformed to ’${name}’.
# Each script should begin:
#
# !INSTALLSHELLPATH −p
#
scripts="a b c"
#
# Transform each script
#
for i in ${scripts}
do
    sed −e "s|INSTALLSHELLPATH|${Pshell}|" < ${i}.source > ${i}
done
```

EXAMPLES

1. Execute a shell command from a string:

   ```
   sh −c "cat myfile"
   ```

2. Execute a shell script from a file in the current directory:

```
sh my_shell_cmds
```

RATIONALE

The *sh* utility and the *set* special built-in utility share a common set of options.

The KornShell ignores the contents of *IFS* upon entry to the script. A conforming application cannot rely on importing *IFS*. One justification for this, beyond security considerations, is to assist possible future shell compilers. Allowing *IFS* to be imported from the environment prevents many optimizations that might otherwise be performed via dataflow analysis of the script itself.

The text in the STDIN section about non-blocking reads concerns an instance of *sh* that has been invoked, probably by a C-language program, with standard input that has been opened using the O_NONBLOCK flag; see *open*( ) in the System Interfaces volume of IEEE Std 1003.1-200x. If the shell did not reset this flag, it would immediately terminate because no input data would be available yet and that would be considered the same as end-of-file.

The options associated with a *restricted shell* (command name *rsh* and the **−r** option) were excluded because the standard developers considered that the implied level of security could not be achieved and they did not want to raise false expectations.

On systems that support set-user-ID scripts, a historical trapdoor has been to link a script to the name **−i**. When it is called by a sequence such as

```
sh −
```

or by:

```
#! usr/bin/sh −
```

the historical systems have assumed that no option letters follow. Thus, this volume of IEEE Std 1003.1-200x allows the single hyphen to mark the end of the options, in addition to the use of the regular "−−" argument, because it was considered that the older practice was so pervasive. An alternative approach is taken by the KornShell, where real and effective user/group IDs must match for an interactive shell; this behavior is specifically allowed by this volume of IEEE Std 1003.1-200x.

Note: There are other problems with set-user-ID scripts that the two approaches described here do not resolve.

The initialization process for the history file can be dependent on the system start-up files, in that they may contain commands that effectively preempt the user's settings of *HISTFILE* and *HISTSIZE*. For example, function definition commands are recorded in the history file, unless the *set* **−o** *nolog* option is set. If the system administrator includes function definitions in some system start-up file called before the *ENV* file, the history file is initialized before the user gets a chance to influence its characteristics.) In some historical shells, the history file is initialized just after the *ENV* file has been processed. Therefore, it is implementation-defined whether changes made to *HISTFILE* after the history file has been initialized are effective.

The default messages for the various *MAIL*-related messages are unspecified because they vary across implementations. Typical messages are:

```
"you have mail\n"
```

or:

```
"you have new mail\n"
```

It is important that the descriptions of command line editing refer to the same shell as that in IEEE Std 1003.1-200x so that interactive users can also be application programmers without having to deal with programmatic differences in their two environments. It is also essential that the utility name *sh* be specified because this explicit utility name is too firmly rooted in historical practice of application programs for it to change.

Consideration was given to mandating a diagnostic message when attempting to set *vi*-mode on terminals that do not support command line editing. However, it is not historical practice for the shell to be cognizant of all terminal types and thus be able to detect inappropriate terminals in all cases. Implementations are encouraged to supply diagnostics in this case whenever possible, rather than leaving the user in a state where editing commands work incorrectly.

In early proposals, the KornShell-derived *emacs* mode of command line editing was included, even though the *emacs* editor itself was not. The community of *emacs* proponents was adamant that the full *emacs* editor not be included in earlier versions of IEEE Std 1003.1 because they were concerned that an attempt to standardize this very powerful environment would encourage vendors to ship versions conforming strictly to earlier versions of IEEE Std 1003.1, but lacking the extensibility required by the community. The author of the original *emacs* program also expressed his desire to omit the program. Furthermore, there were a number of historical systems that did not include *emacs*, or included it without supporting it, but there were very few that did not include and support *vi*. The shell *emacs* command line editing mode was finally omitted from earlier versions of IEEE Std 1003.1 because it became apparent that the KornShell version and the editor being distributed with the GNU system had diverged in some respects. The author of *emacs* requested that the POSIX *emacs* mode either be deleted or have a significant number of unspecified conditions. Although the KornShell author agreed to consider changes to bring the shell into alignment, the standard developers decided to defer specification at this time, rather than attempting to agree on a specific subset of *emacs* late within the development of earlier versions of IEEE Std 1003.1. At the time, it was assumed that convergence on an acceptable definition would occur for a subsequent draft, but that has not happened, and there appears to be no impetus to do so. In any case, implementations are free to offer additional command line editing modes based on the exact models of editors their users are most comfortable with.

Early proposals had the following list entry in **vi Line Editing Insert Mode** (on page 3053):

\	If followed by the *erase* or *kill* character, that character shall be inserted into the input line. Otherwise, the backslash itself shall be inserted into the input line.

However, this is not actually a feature of *sh* command line editing insert mode, but one of some historical terminal line drivers. Some conforming implementations continue to do this when the *stty* **iexten** flag is set.

FUTURE DIRECTIONS

None.

SEE ALSO

cd, *echo*, *pwd*, *test*, *umask*, the System Interfaces volume of IEEE Std 1003.1-200x, *dup*( ), *exec*, *exit*( ), *fork*( ), *pipe*( ), *signal*( ), *system*( ), *ulimit*( ), *umask*( ), *wait*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Text is added to the DESCRIPTION for the Large File Summit proposal.

Issue 6

The Open Group Corrigendum U029/2 is applied, correcting the second SYNOPSIS.

The Open Group Corrigendum U027/3 is applied, correcting a typographical error.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The option letters derived from the set special built-in are also accepted with a leading plus sign (`'+'`).
- Large file extensions are added:
  - Pathname expansion does not fail due to the size of a file.
  - Shell input and output redirections have an implementation-defined offset maximum that is established in the open file description.

In the ENVIRONMENT VARIABLES section, the text ‘‘user’s home directory’’ is updated to ‘‘directory referred to by the *HOME* environment variable’’.

Descriptions for the *ENV* and *PWD* environment variables are included to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

sleep — suspend execution for an interval

SYNOPSIS

`sleep` *`time`*

DESCRIPTION

The *sleep* utility shall suspend execution for at least the integral number of seconds specified by the *time* operand.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

time	A non-negative decimal integer specifying the number of seconds for which to suspend execution.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *sleep*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If the *sleep* utility receives a SIGALRM signal, one of the following actions shall be taken:

1. Terminate normally with a zero exit status.
2. Effectively ignore the signal.
3. Provide the default behavior for signals described in the ASYNCHRONOUS EVENTS section of Section 1.11 (on page 2221). This could include terminating with a non-zero exit status.

The *sleep* utility shall take the standard action for all other signals.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The execution was successfully suspended for at least *time* seconds, or a SIGALRM signal was received. See the ASYNCHRONOUS EVENTS section.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

The *sleep* utility can be used to execute a command after a certain amount of time, as in:

```
(sleep 105; command) &
```

or to execute a command every so often, as in:

```
while true
do
    command
    sleep 37
done
```

RATIONALE

The exit status is allowed to be zero when *sleep* is interrupted by the SIGALRM signal because most implementations of this utility rely on the arrival of that signal to notify them that the requested finishing time has been successfully attained. Such implementations thus do not distinguish this situation from the successful completion case. Other implementations are allowed to catch the signal and go back to sleep until the requested time expires or to provide the normal signal termination procedures.

As with all other utilities that take integral operands and do not specify subranges of allowed values, *sleep* is required by this volume of IEEE Std 1003.1-200x to deal with *time* requests of up to 2 147 483 647 seconds. This may mean that some implementations have to make multiple calls to the delay mechanism of the underlying operating system if its argument range is less than this.

FUTURE DIRECTIONS

None.

SEE ALSO

wait, the System Interfaces volume of IEEE Std 1003.1-200x, *alarm*( ), *sleep*( )

CHANGE HISTORY

First released in Issue 2.

NAME

sort — sort, merge, or sequence check text files

SYNOPSIS

```
sort [−m][−o output][−bdfinru][−t char][−k keydef]... [file...]

sort −c [−bdfinru][−t char][−k keydef][file]
```

DESCRIPTION

The *sort* utility shall perform one of the following functions:

1. Sort lines of all the named files together and write the result to the specified output.
2. Merge lines of all the named (presorted) files together and write the result to the specified output.
3. Check that a single input file is correctly presorted.

Comparisons shall be based on one or more sort keys extracted from each line of input (or, if no sort keys are specified, the entire line up to, but not including, the terminating <newline>), and shall be performed using the collating sequence of the current locale.

OPTIONS

The *sort* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, and the **−k** *keydef* option should follow the **−b**, **−d**, **−f**, **−i**, **−n**, and **−r** options.

The following options shall be supported:

−c	Check that the single input file is ordered as specified by the arguments and the collating sequence of the current locale. No output shall be produced; only the exit code shall be affected.
−m	Merge only; the input file shall be assumed to be already sorted.
−o *output*	Specify the name of an output file to be used instead of the standard output. This file can be the same as one of the input *file*s.
−u	Unique: suppress all but one in each set of lines having equal keys. If used with the **−c** option, check that there are no lines with duplicate keys, in addition to checking that the input file is sorted.

The following options shall override the default ordering rules. When ordering options appear independent of any key field specifications, the requested field ordering rules shall be applied globally to all sort keys. When attached to a specific key (see **−k**), the specified ordering options shall override all global ordering options for that key.

−d	Specify that only <blank>s and alphanumeric characters, according to the current setting of *LC_CTYPE*, shall be significant in comparisons. The behavior is undefined for a sort key to which **−i** or **−n** also applies.
−f	Consider all lowercase characters that have uppercase equivalents, according to the current setting of *LC_CTYPE*, to be the uppercase equivalent for the purposes of comparison.
−i	Ignore all characters that are non-printable, according to the current setting of *LC_CTYPE*.
−n	Restrict the sort key to an initial numeric string, consisting of optional <blank>s, optional minus sign, and zero or more digits with an optional radix character and thousands separators (as defined in the current locale), which shall be sorted by

arithmetic value. An empty digit string shall be treated as zero. Leading zeros and signs on zeros shall not affect ordering.

−r Reverse the sense of comparisons.

The treatment of field separators can be altered using the options:

−b Ignore leading <blank>s when determining the starting and ending positions of a restricted sort key. If the **−b** option is specified before the first **−k** option, it shall be applied to all **−k** options. Otherwise, the **−b** option can be attached independently to each **−k** *field_start* or *field_end* option-argument (see below).

−t *char* Use *char* as the field separator character; *char* shall not be considered to be part of a field (although it can be included in a sort key). Each occurrence of *char* shall be significant (for example, <*char*><*char*> delimits an empty field). If **−t** is not specified, <blank>s shall be used as default field separators; each maximal non-empty sequence of <blank>s that follows a non-<blank> shall be a field separator.

Sort keys can be specified using the options:

−k *keydef* The *keydef* argument is a restricted sort key field definition. The format of this definition is:

`field_start[type][,field_end[type]]`

where *field_start* and *field_end* define a key field restricted to a portion of the line (see the EXTENDED DESCRIPTION section), and *type* is a modifier from the list of characters `'b'`, `'d'`, `'f'`, `'i'`, `'n'`, `'r'`. The `'b'` modifier shall behave like the **−b** option, but shall apply only to the *field_start* or *field_end* to which it is attached. |
The other modifiers shall behave like the corresponding options, but shall apply only to the key field to which they are attached; they shall have this effect if specified with *field_start*, *field_end*, or both. If any modifier is attached to a *field_start* or to a *field_end*, no option shall apply to either. Implementations shall support at least nine occurrences of the **−k** option, which shall be significant in command line order. If no **−k** option is specified, a default sort key of the entire line shall be used.

When there are multiple key fields, later keys shall be compared only after all earlier keys compare equal. Except when the **−u** option is specified, lines that otherwise compare equal shall be ordered as if none of the options **−d**, **−f**, **−i**, **−n**, or **−k** were present (but with **−r** still in effect, if it was specified) and with all bytes in the lines significant to the comparison. The order in which lines that still compare equal are written is unspecified.

OPERANDS

The following operand shall be supported:

file A pathname of a file to be sorted, merged, or checked. If no *file* operands are specified, or if a *file* operand is `'−'`, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'−'`. See the INPUT FILES section.

INPUT FILES

The input files shall be text files, except that the *sort* utility shall add a <newline> to the end of a file ending with an incomplete last line.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *sort*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for ordering rules.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classification for the −**b**, −**d**, −**f**, −**i**, and −**n** options.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_NUMERIC	Determine the locale for the definition of the radix character and thousands separator for the −**n** option.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Unless the −**o** or −**c** options are in effect, the standard output shall contain the sorted input.

STDERR

The standard error shall be used for diagnostic messages. A warning message about correcting an incomplete last line of an input file may be generated, but need not affect the final exit status.

OUTPUT FILES

If the −**o** option is in effect, the sorted input shall be written to the file *output*.

EXTENDED DESCRIPTION

The notation:

```
−k field_start[type][,field_end[type]]
```

shall define a key field that begins at *field_start* and ends at *field_end* inclusive, unless *field_start* falls beyond the end of the line or after *field_end*, in which case the key field is empty. A missing *field_end* shall mean the last character of the line.

A field comprises a maximal sequence of non-separating characters and, in the absence of option −**t**, any preceding field separator.

The *field_start* portion of the *keydef* option-argument shall have the form:

```
field_number[.first_character]
```

Fields and characters within fields shall be numbered starting with 1. The *field_number* and *first_character* pieces, interpreted as positive decimal integers, shall specify the first character to be used as part of a sort key. If .*first_character* is omitted, it shall refer to the first character of the field.

The *field_end* portion of the *keydef* option-argument shall have the form:

```
field_number[.last_character]
```

The *field_number* shall be as described above for *field_start*. The *last_character* piece, interpreted as a non-negative decimal integer, shall specify the last character to be used as part of the sort key. If *last_character* evaluates to zero or *.last_character* is omitted, it shall refer to the last character of the field specified by *field_number*.

If the −**b** option or **b** type modifier is in effect, characters within a field shall be counted from the first non-<blank> in the field. (This shall apply separately to *first_character* and *last_character*.)

EXIT STATUS

The following exit values shall be returned:

0	All input files were output successfully, or −**c** was specified and the input file was correctly sorted.
1	Under the −**c** option, the file was not ordered as specified, or if the −**c** and −**u** options were both specified, two input lines were found with equal keys.
>1	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The default value for −**t**, <blank>, has different properties from, for example, −**t**"<space>". If a line contains:

```
<space><space>foo
```

the following treatment would occur with default separation as opposed to specifically selecting a <space>:

Field	**Default**	**−t "<space>"**
1	<space><space>foo	*empty*
2	*empty*	*empty*
3	*empty*	foo

The leading field separator itself is included in a field when −**t** is not used. For example, this command returns an exit status of zero, meaning the input was already sorted:

```
sort −c −k 2 <<eof
y<tab>b
x<space>a
eof
```

(assuming that a <tab> precedes the <space> in the current collating sequence). The field separator is not included in a field when it is explicitly set via −**t**. This is historical practice and allows usage such as:

```
sort −t "|" −k 2n <<eof
Atlanta|425022|Georgia
Birmingham|284413|Alabama
Columbia|100385|South Carolina
eof
```

where the second field can be correctly sorted numerically without regard to the non-numeric field separator.

The wording in the OPTIONS section clarifies that the −**b**, −**d**, −**f**, −**i**, −**n**, and −**r** options have to come before the first sort key specified if they are intended to apply to all specified keys. The way it is described in this volume of IEEE Std 1003.1-200x matches historical practice, not historical documentation. The results are unspecified if these options are specified after a −**k** option.

The −**f** option might not work as expected in locales where there is not a one-to-one mapping between an uppercase and a lowercase letter.

EXAMPLES

1. The following command sorts the contents of **infile** with the second field as the sort key:

```
sort −k 2,2 infile
```

2. The following command sorts, in reverse order, the contents of **infile1** and **infile2**, placing the output in **outfile** and using the second character of the second field as the sort key (assuming that the first character of the second field is the field separator):

```
sort −r −o outfile −k 2.2,2.2 infile1 infile2
```

3. The following command sorts the contents of **infile1** and **infile2** using the second non-<blank> of the second field as the sort key:

```
sort −k 2.2b,2.2b infile1 infile2
```

4. The following command prints the System V password file (user database) sorted by the numeric user ID (the third colon-separated field):

```
sort −t : −k 3,3n /etc/passwd
```

5. The following command prints the lines of the already sorted file **infile**, suppressing all but one occurrence of lines having the same third field:

```
sort −um −k 3.1,3.0 infile
```

RATIONALE

Examples in some historical documentation state that options −**um** with one input file keep the first in each set of lines with equal keys. This behavior was deemed to be an implementation artifact and was not standardized.

The −**z** option was omitted; it is not standard practice on most systems and is inconsistent with using *sort* to sort several files individually and then merge them together. The text concerning −**z** in historical documentation appeared to require implementations to determine the proper buffer length during the sort phase of operation, but not during the merge.

The −**y** option was omitted because of non-portability. The −**M** option, present in System V, was omitted because of non-portability in international usage.

An undocumented −**T** option exists in some implementations. It is used to specify a directory for intermediate files. Implementations are encouraged to support the use of the *TMPDIR* environment variable instead of adding an option to support this functionality.

The −**k** option was added to satisfy two objections. First, the zero-based counting used by *sort* is not consistent with other utility conventions. Second, it did not meet syntax guideline requirements.

Historical documentation indicates that ‘‘setting −**n** implies −**b**’’. The description of −**n** already states that optional leading <blank>s are tolerated in doing the comparison. If −**b** is enabled, rather than implied, by **−n**, this has unusual side effects. When a character offset is used in a column of numbers (for example, to sort modulo 100), that offset is measured relative to the most significant digit, not to the column. Based upon a recommendation from the author of the original *sort* utility, the **−b** implication has been omitted from this volume of IEEE Std 1003.1-200x, and an application wishing to achieve the previously mentioned side effects has to code the **−b** flag explicitly.

FUTURE DIRECTIONS

None.

SEE ALSO

comm, *join*, *uniq*, the System Interfaces volume of IEEE Std 1003.1-200x, *toupper*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

IEEE PASC Interpretation 1003.2 #174 is applied, updating the DESCRIPTION of comparisons.

IEEE PASC Interpretation 1003.2 #168 is applied.

split

NAME

split — split files into pieces

SYNOPSIS

UP
```
split [−l line_count][−a suffix_length][file[name]]

split −b n[k|m][−a suffix_length][file[name]]
```

DESCRIPTION

The *split* utility shall read an input file and write one or more output files. The default size of each output file shall be 1 000 lines. The size of the output files can be modified by specification of the −**b** or −**l** options. Each output file shall be created with a unique suffix. The suffix shall consist of exactly *suffix_length* lowercase letters from the POSIX locale. The letters of the suffix shall be used as if they were a base-26 digit system, with the first suffix to be created consisting of all `'a'` characters, the second with a `'b'` replacing the last `'a'`, and so on, until a name of all `'z'` characters is created. By default, the names of the output files shall be `'x'`, followed by a two-character suffix from the character set as described above, starting with `"aa"`, `"ab"`, `"ac"`, and so on, and continuing until the suffix `"zz"`, for a maximum of 676 files.

If the number of files required exceeds the maximum allowed by the suffix length provided, such that the last allowable file would be larger than the requested size, the *split* utility shall fail after creating the last file with a valid suffix; *split* shall not delete the files it created with valid suffixes. If the file limit is not exceeded, the last file created shall contain the remainder of the input file, and may be smaller than the requested size.

OPTIONS

The *split* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−**a** *suffix_length*	Use *suffix_length* letters to form the suffix portion of the filenames of the split file. If −**a** is not specified, the default suffix length shall be two. If the sum of the *name* operand and the *suffix_length* option-argument would create a filename exceeding {NAME_MAX} bytes, an error shall result; *split* shall exit with a diagnostic message and no files shall be created.
−**b** *n*	Split a file into pieces *n* bytes in size.
−**b** *n***k**	Split a file into pieces *n**1024 bytes in size.
−**b** *n***m**	Split a file into pieces *n**1 048 576 bytes in size.
−**l** *line_count*	Specify the number of lines in each resulting file piece. The *line_count* argument is an unsigned decimal integer. The default is 1 000. If the input does not end with a <newline>, the partial line shall be included in the last output file.

OPERANDS

The following operands shall be supported:

file	The pathname of the ordinary file to be split. If no input file is given or *file* is `'−'`, the standard input shall be used.
name	The prefix to be used for each of the files resulting from the split operation. If no *name* argument is given, `'x'` shall be used as the prefix of the output files. The combined length of the basename of *prefix* and *suffix_length* cannot exceed {NAME_MAX} bytes. See the OPTIONS section.

STDIN

See the INPUT FILES section.

INPUT FILES

Any file can be used as input.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *split*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output files contain portions of the original input file; otherwise, unchanged.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

In the following examples **foo** is a text file that contains 5 000 lines.

1. Create five files, **xaa**, **xab**, **xac**, **xad**, and **xae**:

   ```
   split foo
   ```

2. Create five files, but the suffixed portion of the created files consists of three letters, **xaaa**, **xaab**, **xaac**, **xaad**, and **xaae**:

   ```
   split −a 3 foo
   ```

3. Create three files with four-letter suffixes and a supplied prefix, **bar_aaaa**, **bar_aaab**, and **bar_aaac**:

   ```
   split −a 4 −l 2000 foo bar_
   ```

4. Create as many files as are necessary to contain at most 20*1 024 bytes, each with the default prefix of **x** and a five-letter suffix:

   ```
   split −a 5 −b 20k foo
   ```

RATIONALE

The −**b** option was added to provide a mechanism for splitting files other than by lines. While most uses of the −**b** option are for transmitting files over networks, some believed it would have additional uses.

The −**a** option was added to overcome the limitation of being able to create only 676 files.

Consideration was given to deleting this utility, using the rationale that the function provided by this utility is available via the *csplit* utility (see *csplit* (on page 2480)). Upon reconsideration of the purpose of the User Portability Extension, it was decided to retain both this utility and the *csplit* utility because users use both utilities and have historical expectations of their behavior. Furthermore, the splitting on byte boundaries in *split* cannot be duplicated with the historical *csplit*.

The text ''*split* shall not delete the files it created with valid suffixes'' would normally be assumed, but since the related utility, *csplit*, does delete files under some circumstances, the historical behavior of *split* is made explicit to avoid misinterpretation.

FUTURE DIRECTIONS

None.

SEE ALSO

csplit

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

The obsolescent SYNOPSIS is removed.

NAME

strings — find printable strings in files

SYNOPSIS

UP `strings [−a][−t format][−n number][file...]`

DESCRIPTION

The *strings* utility shall look for printable strings in regular files and shall write those strings to standard output. A printable string is any sequence of four (by default) or more printable characters terminated by a <newline> or NUL character. Additional implementation-defined strings may be written; see *localedef*.

OPTIONS

The *strings* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Scan files in their entirety. If **−a** is not specified, it is implementation-defined what portion of each file is scanned for strings.
−n *number*	Specify the minimum string length, where the *number* argument is a positive decimal integer. The default shall be 4.
−t *format*	Write each string preceded by its byte offset from the start of the file. The format shall be dependent on the single character used as the *format* option-argument:

d	The offset shall be written in decimal.
o	The offset shall be written in octal.
x	The offset shall be written in hexadecimal.

OPERANDS

The following operand shall be supported:

file	A pathname of a regular file to be used as input. If no *file* operand is specified, the *strings* utility shall read from the standard input.

STDIN

See the INPUT FILES section.

INPUT FILES

The input files named by the utility arguments or the standard input shall be regular files of any format.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *strings*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and to identify printable strings.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Strings found shall be written to the standard output, one per line.

When the **−t** option is not specified, the format of the output shall be:

```
"%s", <string>
```

With the **−t o** option, the format of the output shall be:

```
"%o %s", <byte offset>, <string>
```

With the **−t x** option, the format of the output shall be:

```
"%x %s", <byte offset>, <string>
```

With the **−t d** option, the format of the output shall be:

```
"%d %s", <byte offset>, <string>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

By default the data area (as opposed to the text, ‘‘bss’’ or header areas) of a binary executable file is scanned. Implementations document which areas are scanned.

Some historical implementations do not require NUL or <newline> terminators for strings to permit those languages that do not use NUL as a string terminator to have their strings written.

EXAMPLES

None.

RATIONALE

Apart from rationalizing the option syntax and slight difficulties with object and executable binary files, *strings* is specified to match historical practice closely. The **−a** and **−n** options were introduced to replace the non-conforming – and −*number* options.

The **−o** option historically means different things on different implementations. Some use it to mean ‘‘*offset* in decimal’’, while others use it as ‘‘*offset* in octal’’. Instead of trying to decide which way would be least objectionable, the **−t** option was added. It was originally named **−O** to mean ‘‘offset’’, but was changed to **−t** to be consistent with *od*.

The ISO C standard function *isprint*( ) is restricted to a domain of **unsigned char**. This volume of IEEE Std 1003.1-200x requires implementations to write strings as defined by the current locale.

FUTURE DIRECTIONS

None.

SEE ALSO

nm

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The obsolescent SYNOPSIS is removed.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

strip — remove unnecessary information from executable files (**DEVELOPMENT**)

SYNOPSIS

SD
```
strip file...
```

DESCRIPTION

The *strip* utility shall remove from executable files named by the *file* operands any information the implementor deems unnecessary for execution of those files. The nature of that information is unspecified. The effect of *strip* shall be similar to the use of the –**s** option to *c99* or *fort77*.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

file	A pathname referring to an executable file.

STDIN

Not used.

INPUT FILES

The input files shall be in the form of executable files successfully produced by any compiler defined by this volume of IEEE Std 1003.1-200x.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *strip*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The *strip* utility shall produce executable files of unspecified format.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

Historically, this utility has been used to remove the symbol table from an executable file. It was included since it is known that the amount of symbolic information can amount to several megabytes; the ability to remove it in a portable manner was deemed important, especially for smaller systems.

The behavior of *strip* is said to be the same as the **–s** option to a compiler. While the end result is essentially the same, it is not required to be identical.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *c99*, *fort77*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the Software Development Utilities option.

NAME

stty — set the options for a terminal

SYNOPSIS

```
stty [ −a| −g]

stty operands
```

DESCRIPTION

The *stty* utility shall set or report on terminal I/O characteristics for the device that is its standard input. Without options or operands specified, it shall report the settings of certain characteristics, usually those that differ from implementation-defined defaults. Otherwise, it shall modify the terminal state according to the specified operands. Detailed information about the modes listed in the first five groups below are described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface. Operands in the Combination Modes group (see **Combination Modes** (on page 3087)) are implemented using operands in the previous groups. Some combinations of operands are mutually-exclusive on some terminal types; the results of using such combinations are unspecified.

Typical implementations of this utility require a communications line configured to use the **termios** interface defined in the System Interfaces volume of IEEE Std 1003.1-200x. On systems where none of these lines are available, and on lines not currently configured to support the **termios** interface, some of the operands need not affect terminal characteristics.

OPTIONS

The *stty* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Write to standard output all the current settings for the terminal.
−g	Write to standard output all the current settings in an unspecified form that can be used as arguments to another invocation of the *stty* utility on the same system. The form used shall not contain any characters that would require quoting to avoid word expansion by the shell; see Section 2.6 (on page 2238).

OPERANDS

The following operands shall be supported to set the terminal characteristics.

Control Modes

parenb (**−parenb**)	Enable (disable) parity generation and detection. This shall have the effect of setting (not setting) PARENB in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
parodd (**−parodd**)	Select odd (even) parity. This shall have the effect of setting (not setting) PARODD in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
cs5 cs6 cs7 cs8	Select character size, if possible. This shall have the effect of setting CS5, CS6, CS7, and CS8, respectively, in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
number	Set terminal baud rate to the number given, if possible. If the baud rate is set to zero, the modem control lines shall not be longer asserted. This shall have the effect of setting the input and output **termios** baud rate values as defined

in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

ispeed *number* Set terminal input baud rate to the number given, if possible. If the input baud rate is set to zero, the input baud rate shall be specified by the value of the output baud rate. This shall have the effect of setting the input **termios** baud rate values as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

ospeed *number* Set terminal output baud rate to the number given, if possible. If the output baud rate is set to zero, the modem control lines shall no longer be asserted. This shall have the effect of setting the output **termios** baud rate values as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

hupcl (−**hupcl**) Stop asserting modem control lines (do not stop asserting modem control lines) on last close. This shall have the effect of setting (not setting) HUPCL in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

hup (−**hup**) Equivalent to **hupcl**(−**hupcl**).

cstopb (−**cstopb**) Use two (one) stop bits per character. This shall have the effect of setting (not setting) CSTOPB in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

cread (−**cread**) Enable (disable) the receiver. This shall have the effect of setting (not setting) CREAD in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

clocal (−**clocal**) Assume a line without (with) modem control. This shall have the effect of setting (not setting) CLOCAL in the **termios** *c_cflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

It is unspecified whether *stty* shall report an error if an attempt to set a Control Mode fails.

Input Modes

ignbrk (−**ignbrk**) Ignore (do not ignore) break on input. This shall have the effect of setting (not setting) IGNBRK in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

brkint (−**brkint**) Signal (do not signal) INTR on break. This shall have the effect of setting (not setting) BRKINT in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

ignpar (−**ignpar**) Ignore (do not ignore) bytes with parity errors. This shall have the effect of setting (not setting) IGNPAR in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

parmrk (−**parmrk**)

Mark (do not mark) parity errors. This shall have the effect of setting (not setting) PARMRK in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

inpck (**–inpck**) Enable (disable) input parity checking. This shall have the effect of setting (not setting) INPCK in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

istrip (**–istrip**) Strip (do not strip) input characters to seven bits. This shall have the effect of setting (not setting) ISTRIP in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

inlcr (**–inlcr**) Map (do not map) NL to CR on input. This shall have the effect of setting (not setting) INLCR in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

igncr (–igncr) Ignore (do not ignore) CR on input. This shall have the effect of setting (not setting) IGNCR in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

icrnl (**–icrnl**) Map (do not map) CR to NL on input. This shall have the effect of setting (not setting) ICRNL in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

ixon (**–ixon**) Enable (disable) START/STOP output control. Output from the system is stopped when the system receives STOP and started when the system receives START. This shall have the effect of setting (not setting) IXON in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

XSI **ixany** (**–ixany**) Allow any character to restart output. This shall have the effect of setting (not setting) IXANY in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

ixoff (**–ixoff**) Request that the system send (not send) STOP characters when the input queue is nearly full and START characters to resume data transmission. This shall have the effect of setting (not setting) IXOFF in the **termios** *c_iflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

Output Modes

opost (**–opost**) Post-process output (do not post-process output; ignore all other output modes). This shall have the effect of setting (not setting) OPOST in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

XSI **ocrnl** (**–ocrnl**) Map (do not map) CR to NL on output This shall have the effect of setting (not setting) OCRNL in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

onocr (**–onocr**) Do not (do) output CR at column zero. This shall have the effect of setting (not setting) ONOCR in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

onlret (**–onlret**) The terminal newline key performs (does not perform) the CR function. This shall have the effect of setting (not setting) ONLRET in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

ofill (−**ofill**)	Use fill characters (use timing) for delays. This shall have the effect of setting (not setting) OFILL in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
ofdel (−**ofdel**)	Fill characters are DELs (NULs). This shall have the effect of setting (not setting) OFDEL in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
cr0 cr1 cr2 cr3	Select the style of delay for CRs. This shall have the effect of setting CRDLY to CR0, CR1, CR2, or CR3, respectively, in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
nl0 nl1	Select the style of delay for NL. This has the effect of setting NLDLY to NL0 or NL1, respectively, in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
tab0 tab1 tab2 tab3	Select the style of delay for horizontal tabs. This shall have the effect of setting TABDLY to TAB0, TAB1, TAB2, or TAB3, respectively, in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface. Note that TAB3 has the effect of expanding <tab>s to <space>s.
tabs (−**tabs**)	Synonym for **tab0** (**tab3**).
bs0 bs1	Select the style of delay for backspaces. This shall have the effect of setting BSDLY to BS0 or BS1, respectively, in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
ff0 ff1	Select the style of delay for form-feeds. This shall have the effect of setting FFDLY to FF0 or FF1, respectively, in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
vt0 vt1	Select the style of delay for vertical-tabs. This shall have the effect of setting VTDLY to VT0 or VT1, respectively, in the **termios** *c_oflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

Local Modes

isig (−**isig**)	Enable (disable) the checking of characters against the special control characters INTR, QUIT, and SUSP. This shall have the effect of setting (not setting) ISIG in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
icanon (−**icanon**)	Enable (disable) canonical input (ERASE and KILL processing). This shall have the effect of setting (not setting) ICANON in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
iexten (−**iexten**)	Enable (disable) any implementation-defined special control characters not currently controlled by **icanon**, **isig**, **ixon**, or **ixoff**. This shall have the effect of setting (not setting) IEXTEN in the **termios** *c_lflag* field, as defined in the Base

Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

echo (−**echo**) Echo back (do not echo back) every character typed. This shall have the effect of setting (not setting) ECHO in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

echoe (−**echoe**) The ERASE character visually erases (does not erase) the last character in the current line from the display, if possible. This shall have the effect of setting (not setting) ECHOE in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

echok (−**echok**) Echo (do not echo) NL after KILL character. This shall have the effect of setting (not setting) ECHOK in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

echonl (−**echonl**) Echo (do not echo) NL, even if **echo** is disabled. This shall have the effect of setting (not setting) ECHONL in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

noflsh (−**noflsh**) Disable (enable) flush after INTR, QUIT, SUSP. This shall have the effect of setting (not setting) NOFLSH in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

tostop (−**tostop**) Send SIGTTOU for background output. This shall have the effect of setting (not setting) TOSTOP in the **termios** *c_lflag* field, as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.

Special Control Character Assignments

<control>-character string

Set *<control>-character* to *string*. If *<control>-character* is one of the character sequences in the first column of the following table, the corresponding Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface control character from the second column shall be recognized. This has the effect of setting the corresponding element of the **termios** *c_cc* array (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 13, Headers, **<termios.h>**).

Table 4-19 Control Character Names in *stty*

Control Character	c_cc Subscript	Description
eof	VEOF	EOF character
eol	VEOL	EOL character
erase	VERASE	ERASE character
intr	VINTR	INTR character
kill	VKILL	KILL character
quit	VQUIT	QUIT character
susp	VSUSP	SUSP character
start	VSTART	START character
stop	VSTOP	STOP character

If *string* is a single character, the control character shall be set to that character. If *string* is the two-character sequence `"^-"` or the string *undef*, the control character shall be set to _POSIX_VDISABLE , if it is in effect for the device; if _POSIX_VDISABLE is not in effect for the device, it shall be treated as an error. In the POSIX locale, if *string* is a two-character sequence beginning with circumflex (`'^'`), and the second character is one of those listed in the `"^c"` column of the following table, the control character shall be set to the corresponding character value in the Value column of the table.

Table 4-20 Circumflex Control Characters in *stty*

^c	Value	^c	Value	^c	Value
`a, A`	<SOH>	`l, L`	<FF>	`w, W`	<ETB>
`b, B`	<STX>	`m, M`	<CR>	`x, X`	<CAN>
`c, C`	<ETX>	`n, N`	<SO>	`y, Y`	<EM>
`d, D`	<EOT>	`o, O`	<SI>	`z, Z`	<SUB>
`e, E`	<ENQ>	`p, P`	<DLE>	`[`	<ESC>
`f, F`	<ACK>	`q, Q`	<DC1>	`\`	<FS>
`g, G`	<BEL>	`r, R`	<DC2>	`]`	<GS>
`h, H`	<BS>	`s, S`	<DC3>	`^`	<RS>
`i, I`	<HT>	`t, T`	<DC4>	`_`	<US>
`j, J`	<LF>	`u, U`	<NAK>	`?`	<DEL>
`k, K`	<VT>	`v, V`	<SYN>		

min *number*

Set the value of MIN to *number*. MIN is used in non-canonical mode input processing (**icanon**).

time *number*

Set the value of TIME to *number*. TIME is used in non-canonical mode input processing (**icanon**).

Combination Modes

saved settings

Set the current terminal characteristics to the saved settings produced by the **−g** option.

evenp or **parity**

Enable **parenb** and **cs7**; disable **parodd**.

oddp

Enable **parenb**, **cs7**, and **parodd**.

−**parity**, −**evenp**, or −**oddp**
Disable **parenb**, and set **cs8**.

XSI **raw** (−**raw** or **cooked**)
Enable (disable) raw input and output. Raw mode shall be equivalent to setting:

```
stty cs8 erase ^- kill ^- intr ^- \
    quit ^- eof ^- eol ^- -post -inpck
```

nl (−**nl**)
Enable (disable) **icrnl**. In addition, −**nl** unsets **inlcr** and **igncr**.

ek Reset ERASE and KILL characters back to system defaults.

sane Reset all modes to some reasonable, unspecified, values.

STDIN

Although no input is read from standard input, standard input shall be used to get the current terminal I/O characteristics and to set new terminal I/O characteristics.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *stty*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE This variable determines the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments) and which characters are in the class **print**.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If operands are specified, no output shall be produced.

If the −**g** option is specified, *stty* shall write to standard output the current settings in a form that can be used as arguments to another instance of *stty* on the same system.

If the −**a** option is specified, all of the information as described in the OPERANDS section shall be written to standard output. Unless otherwise specified, this information shall be written as <space>-separated tokens in an unspecified format, on one or more lines, with an unspecified number of tokens per line. Additional information may be written.

If no options or operands are specified, an unspecified subset of the information written for the −**a** option shall be written.

If speed information is written as part of the default output, or if the −**a** option is specified and if the terminal input speed and output speed are the same, the speed information shall be written as follows:

```
"speed %d baud;", <speed>
```

Otherwise, speeds shall be written as:

```
"ispeed %d baud; ospeed %d baud;", <ispeed>, <ospeed>
```

In locales other than the POSIX locale, the word **baud** may be changed to something more appropriate in those locales.

If control characters are written as part of the default output, or if the −**a** option is specified, control characters shall be written as:

```
"%s = %s;", <control-character name>, <value>
```

where <*value*> is either the character, or some visual representation of the character if it is non-printable, or the string *undef* if the character is disabled.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The terminal options were read or set successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The −**g** flag is designed to facilitate the saving and restoring of terminal state from the shell level. For example, a program may:

```
saveterm="$(stty −g)"       # save terminal state
stty (new settings)         # set new state
...                        # ...
stty $saveterm              # restore terminal state
```

Since the format is unspecified, the saved value is not portable across systems.

Since the −**a** format is so loosely specified, scripts that save and restore terminal settings should use the −**g** option.

EXAMPLES

None.

RATIONALE

The original *stty* description was taken directly from System V and reflected the System V terminal driver **termio**. It has been modified to correspond to the terminal driver **termios**.

Output modes are specified only for XSI-conformant systems. All implementations are expected to provide *stty* operands corresponding to all of the output modes they support.

The *stty* utility is primarily used to tailor the user interface of the terminal, such as selecting the preferred ERASE and KILL characters. As an application programming utility, *stty* can be used within shell scripts to alter the terminal settings for the duration of the script.

The **termios** section states that individual disabling of control characters is possible through the option _POSIX_VDISABLE. If enabled, two conventions currently exist for specifying this: System V uses `"^−"`, and BSD uses *undef*. Both are accepted by *stty* in this volume of IEEE Std 1003.1-200x. The other BSD convention of using the letter `'u'` was rejected because it conflicts with the actual letter `'u'`, which is an acceptable value for a control character.

Early proposals did not specify the mapping of `"^c"` to control characters because the control characters were not specified in the POSIX locale character set description file requirements. The control character set is now specified in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 3, Definitions so the historical mapping is specified. Note that although the mapping corresponds to control-character key assignments on many terminals that use the ISO/IEC 646:1991 standard (or ASCII) character encodings, the mapping specified here is to the control characters, not their keyboard encodings.

Since **termios** supports separate speeds for input and output, two new options were added to specify each distinctly.

Some historical implementations use standard input to get and set terminal characteristics; others use standard output. Since input from a login TTY is usually restricted to the owner while output to a TTY is frequently open to anyone, using standard input provides fewer chances of accidentally (or maliciously) altering the terminal settings of other users. Using standard input also allows *stty* −**a** and *stty* −**g** output to be redirected for later use. Therefore, usage of standard input is required by this volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

The Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 2.

Issue 5

The description of **tabs** is clarified.

FUTURE DIRECTIONS section added.

Issue 6

The legacy items **iuclc**(–**iuclc**), **xcase**, **olcuc**(–**olcuc**), **lcase**(–**lcase**), and **LCASE**(–**LCASE**), are removed.

NAME

tabs — set terminal tabs

SYNOPSIS

```
tabs [ −n| −a| −a2| −c| −c2| −c3| −f| −p| −s| −u][+m[n]] [−T type]

tabs [−T type][ +[n]] n1[,n2,...]
```

DESCRIPTION

The *tabs* utility shall display a series of characters that first clears the hardware terminal tab settings and then initializes the tab stops at the specified positions and optionally adjusts the margin.

The phrase ‘‘tab-stop position *N*’’ shall be taken to mean that, from the start of a line of output, tabbing to position *N* shall cause the next character output to be in the (*N*+1)th column position on that line. The maximum number of tab stops allowed is terminal-dependent.

It need not be possible to implement *tabs* on certain terminals. If the terminal type obtained from the *TERM* environment variable or **−T** option represents such a terminal, an appropriate diagnostic message shall be written to standard error and *tabs* shall exit with a status greater than zero.

OPTIONS

The *tabs* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except for various extensions: the options **−a2**, **−c2**, and **−c3** are multi-character.

The following options shall be supported:

−*n*	Specify repetitive tab stops separated by a uniform number of column positions, *n*, where *n* is a single-digit decimal number. The default usage of *tabs* with no arguments shall be equivalent to **tabs**−8. When **−0** is used, the tab stops shall be cleared and no new ones set.
−a	1,10,16,36,72 Assembler, applicable to some mainframes.
−a2	1,10,16,40,72 Assembler, applicable to some mainframes.
−c	1,8,12,16,20,55 COBOL, normal format.
−c2	1,6,10,14,49 COBOL, compact format (columns 1 to 6 omitted).
−c3	1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67 COBOL compact format (columns 1 to 6 omitted), with more tabs than **−c2**.
−f	1,7,11,15,19,23 FORTRAN
−p	1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61 PL/1
−s	1,10,55 SNOBOL
−u	1,12,20,44 Assembler, applicable to some mainframes.

–T *type*	Indicate the type of terminal. If this option is not supplied and the *TERM* variable is unset or null, an unspecified default terminal type shall be used. The setting of *type* shall take precedence over the value in *TERM*.

OPERANDS

The following operand shall be supported:

*n1***[**,*n2*,. . .**]**	A single command line argument that consists of tab-stop values separated using either commas or <blank>s. The application shall ensure that the tab-stop values are positive decimal integers in strictly ascending order. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. For example, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered to be identical.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tabs*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
	TERM	Determine the terminal type. If this variable is unset or null, and if the **–T** option is not specified, an unspecified default terminal type shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If standard output is a terminal, the appropriate sequence to clear and set the tab stops may be written to standard output in an unspecified format. If standard output is not a terminal, undefined results occur.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility makes use of the terminal's hardware tabs and the *stty tabs* option.

This utility is not recommended for application use.

Some integrated display units might not have escape sequences to set tab stops, but may be set by internal system calls. On these terminals, *tabs* works if standard output is directed to the terminal; if output is directed to another file, however, *tabs* fails.

EXAMPLES

None.

RATIONALE

Consideration was given to having the *tput* utility handle all of the functions described in *tabs*. However, the separate *tabs* utility was retained because it seems more intuitive to use a command named *tabs* than *tput* with a new option. The POSIX Shell and Utilities *tput* does not support setting or clearing tabs, and no known historical version of *tabs* supports the capability of setting arbitrary tab stops.

The System V *tabs* interface is very complex; the version in this volume of IEEE Std 1003.1-200x has a reduced feature list, but many of the features omitted were restored as XSI extensions even though the supported languages and coding styles are primarily historical.

There was considerable sentiment for specifying only a means of resetting the tabs back to a known state—presumably the ''standard'' of tabs every eight positions. The following features were omitted:

- Setting tab stops via the first line in a file, using −−*file*. Since even the SVID has no complete explanation of this feature, it is doubtful that it is in widespread use.

In an early proposal, a **−t** *tablist* option was added for consistency with *expand*; this was later removed when inconsistencies with the historical list of tabs were identified.

Consideration was given to adding a **−p** option that would output the current tab settings so that they could be saved and then later restored. This was not accepted because querying the tab stops of the terminal is not a capability in historical *terminfo* or *termcap* facilities and might not be supported on a wide range of terminals.

FUTURE DIRECTIONS

None.

SEE ALSO

expand, *stty*, *unexpand*

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

tail — copy the last part of a file

SYNOPSIS

`tail [−f][ −c number| −n number][file]`

DESCRIPTION

The *tail* utility shall copy its input file to the standard output beginning at a designated place.

Copying shall begin at the point in the file indicated by the −**c** *number* or −**n** *number* options. The option-argument *number* shall be counted in units of lines or bytes, according to the options −**n** and −**c**. Both line and byte counts start from 1.

Tails relative to the end of the file may be saved in an internal buffer, and thus may be limited in length. Such a buffer, if any, shall be no smaller than {LINE_MAX}*10 bytes.

OPTIONS

The *tail* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−**c** *number* The application shall ensure that the *number* option-argument is a decimal integer whose sign affects the location in the file, measured in bytes, to begin the copying:

Sign	Copying Starts
+	Relative to the beginning of the file.
−	Relative to the end of the file.
none	Relative to the end of the file.

The origin for counting shall be 1; that is, −**c** +1 represents the first byte of the file, −**c** −1 the last.

−**f** If the input file is a regular file or if the *file* operand specifies a FIFO, do not terminate after the last line of the input file has been copied, but read and copy further bytes from the input file when they become available. If no *file* operand is specified and standard input is a pipe, the −**f** option shall be ignored. If the input file is not a FIFO, pipe, or regular file, it is unspecified whether or not the −**f** option shall be ignored.

−**n** *number* This option shall be equivalent to −**c** *number*, except the starting location in the file shall be measured in lines instead of bytes. The origin for counting shall be 1; that is, −**n** +1 represents the first line of the file, −**n** −1 the last.

If neither −**c** nor −**n** is specified, −**n** 10 shall be assumed.

OPERANDS

The following operand shall be supported:

file A pathname of an input file. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

If the −**c** option is specified, the input file can contain arbitrary data; otherwise, the input file shall be a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tail*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The designated portion of the input file shall be written to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The −**c** option should be used with caution when the input is a text file containing multi-byte characters; it may produce output that does not start on a character boundary.

Although the input file to *tail* can be any type, the results might not be what would be expected on some character special device files or on file types not described by the System Interfaces volume of IEEE Std 1003.1-200x. Since this volume of IEEE Std 1003.1-200x does not specify the block size used when doing input, *tail* need not read all of the data from devices that only perform block transfers.

EXAMPLES

The –**f** option can be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail –f fred
```

prints the last ten lines of the file **fred**, followed by any lines that are appended to **fred** between the time *tail* is initiated and killed. As another example, the command:

```
tail –f –c 15 fred
```

prints the last 15 bytes of the file **fred**, followed by any bytes that are appended to **fred** between the time *tail* is initiated and killed.

RATIONALE

This version of *tail* was created to allow conformance to the Utility Syntax Guidelines. The historical –**b** option was omitted because of the general non-portability of block-sized units of text. The –**c** option historically meant ''characters'', but this volume of IEEE Std 1003.1-200x indicates that it means ''bytes''. This was selected to allow reasonable implementations when multi-byte characters are possible; it was not named –**b** to avoid confusion with the historical –**b**.

The origin of counting both lines and bytes is 1, matching all widespread historical implementations.

The restriction on the internal buffer is a compromise between the historical System V implementation of 4 096 bytes and the BSD 32 768 bytes.

The –**f** option has been implemented as a loop that sleeps for 1 second and copies any bytes that are available. This is sufficient, but if more efficient methods of determining when new data are available are developed, implementations are encouraged to use them.

Historical documentation indicates that *tail* ignores the –**f** option if the input file is a pipe (pipe and FIFO on systems that support FIFOs). On BSD-based systems, this has been true; on System V-based systems, this was true when input was taken from standard input, but it did not ignore the –**f** flag if a FIFO was named as the *file* operand. Since the –**f** option is not useful on pipes and all historical implementations ignore –**f** if no *file* operand is specified and standard input is a pipe, this volume of IEEE Std 1003.1-200x requires this behavior. However, since the –**f** option is useful on a FIFO, this volume of IEEE Std 1003.1-200x also requires that if standard input is a FIFO or a FIFO is named, the –**f** option shall not be ignored. Although historical behavior does not ignore the –**f** option for other file types, this is unspecified so that implementations are allowed to ignore the –**f** option if it is known that the file cannot be extended.

This was changed to the current form based on comments noting that –**c** was almost never used without specifying a number and that there was no need to specify –**l** if –**n** *number* was given.

FUTURE DIRECTIONS

None.

SEE ALSO

head

CHANGE HISTORY

First released in Issue 2.

Issue 6

The obsolescent SYNOPSIS lines and associated text are removed.

The normative text is reworded to avoid use of the term ''must'' for application requirements.

NAME

talk — talk to another user

SYNOPSIS

UP

```
talk address [terminal]
```

DESCRIPTION

The *talk* utility is a two-way, screen-oriented communication program.

When first invoked, *talk* shall send a message similar to:

```
Message from <unspecified string>
talk: connection requested by your_address
talk: respond with: talk your_address
```

to the specified *address*. At this point, the recipient of the message can reply by typing:

```
talk your_address
```

Once communication is established, the two parties can type simultaneously, with their output displayed in separate regions of the screen. Characters shall be processed as follows:

- Typing the alert character shall alert the recipient's terminal.
- Typing <control>-L shall cause the sender's screen regions to be refreshed.
- Typing the erase and kill characters shall affect the sender's terminal in the manner described by the **termios** interface in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
- Typing the interrupt or end-of-file characters shall terminate the local *talk* utility. Once the *talk* session has been terminated on one side, the other side of the *talk* session shall be notified that the *talk* session has been terminated and shall be able to do nothing except exit.
- Typing characters from *LC_CTYPE* classifications **print** or **space** shall cause those characters to be sent to the recipient's terminal.
- When and only when the *stty* **iexten** local mode is enabled, the existence and processing of additional special control characters and multi-byte or single-byte functions shall be implementation-defined.
- Typing other non-printable characters shall cause implementation-defined sequences of printable characters to be sent to the recipient's terminal.

Permission to be a recipient of a *talk* message can be denied or granted by use of the *mesg* utility. However, a user's privilege may further constrain the domain of accessibility of other users' terminals. The *talk* utility shall fail when the user lacks the appropriate privileges to perform the requested action.

Certain block-mode terminals do not have all the capabilities necessary to support the simultaneous exchange of messages required for *talk*. When this type of exchange cannot be supported on such terminals, the implementation may support an exchange with reduced levels of simultaneous interaction or it may report an error describing the terminal-related deficiency.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

address	The recipient of the *talk* session. One form of *address* is the <*user name*>, as returned by the *who* utility. Other address formats and how they are handled are unspecified.
terminal	If the recipient is logged in more than once, the *terminal* argument can be used to indicate the appropriate terminal name. If *terminal* is not specified, the *talk* message shall be displayed on one or more accessible terminals in use by the recipient. The format of *terminal* shall be the same as that returned by the *who* utility.

STDIN

Characters read from standard input shall be copied to the recipient's terminal in an unspecified manner. If standard input is not a terminal, talk shall write a diagnostic message and exit with a non-zero status.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *talk*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files). If the recipient's locale does not use an *LC_CTYPE* equivalent to the sender's, the results are undefined.
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
	TERM	Determine the name of the invoker's terminal type. If this variable is unset or null, an unspecified default terminal type shall be used.

ASYNCHRONOUS EVENTS

When the *talk* utility receives a SIGINT signal, the utility shall terminate and exit with a zero status. It shall take the standard action for all other signals.

STDOUT

If standard output is a terminal, characters copied from the recipient's standard input may be written to standard output. Standard output also may be used for diagnostic messages. If standard output is not a terminal, *talk* shall exit with a non-zero status.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred or *talk* was invoked on a terminal incapable of supporting it.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Because the handling of non-printable, non-<space>s is tied to the *stty* description of **iexten**, implementation extensions within the terminal driver can be accessed. For example, some implementations provide line editing functions with certain control character sequences.

EXAMPLES

None.

RATIONALE

The *write* utility was included in this volume of IEEE Std 1003.1-200x since it can be implemented on all terminal types. The *talk* utility, which cannot be implemented on certain terminals, was considered to be a ''better'' communications interface. Both of these programs are in widespread use on historical implementations. Therefore, both utilities have been specified.

All references to networking abilities (*talk*ing to a user on another system) were removed as being outside the scope of this volume of IEEE Std 1003.1-200x.

Historical BSD and System V versions of *talk* terminate both of the conversations when either user breaks out of the session. This can lead to adverse consequences if a user unwittingly continues to enter text that is interpreted by the shell when the other terminates the session. Therefore, the version of *talk* specified by this volume of IEEE Std 1003.1-200x requires both users to terminate their end of the session explicitly.

Only messages sent to the terminal of the invoking user can be internationalized in any way:

- The original ''Message from <*unspecified string*> ...'' message sent to the terminal of the recipient cannot be internationalized because the environment of the recipient is as yet inaccessible to the *talk* utility. The environment of the invoking party is irrelevant.
- Subsequent communication between the two parties cannot be internationalized because the two parties may specify different languages in their environment (and non-portable characters cannot be mapped from one language to another).
- Neither party can be required to communicate in a language other than C and/or the one specified by their environment because unavailable terminal hardware support (for example, fonts) may be required.

The text in the STDOUT section reflects the usage of the verb ''display'' in this section; some *talk* implementations actually use standard output to write to the terminal, but this volume of IEEE Std 1003.1-200x does not require that to be the case.

The format of the terminal name is unspecified, but the descriptions of *ps*, *talk*, *who*, and *write* require that they all use or accept the same format.

The handling of non-printable characters is partially implementation-defined because the details of mapping them to printable sequences is not needed by the user. Historical implementations, for security reasons, disallow the transmission of non-printable characters that may send commands to the other terminal.

FUTURE DIRECTIONS

None.

SEE ALSO

mesg, *who*, *write*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

NAME

tee — duplicate standard input

SYNOPSIS

```
tee [−ai][file...]
```

DESCRIPTION

The *tee* utility shall copy standard input to standard output, making a copy in zero or more files. The *tee* utility shall not buffer output.

If the **−a** option is not specified, output files shall be written (see Section 1.7.1.4 (on page 2204).

OPTIONS

The *tee* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Append the output to the files.
−i	Ignore the SIGINT signal.

OPERANDS

The following operands shall be supported:

file	A pathname of an output file. Processing of at least 13 *file* operands shall be supported.

STDIN

The standard input can be of any type.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tee*:

	LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
	LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
	LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
	LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default, except that if the **−i** option was specified, SIGINT shall be ignored.

STDOUT

The standard output shall be a copy of the standard input.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

If any *file* operands are specified, the standard input shall be copied to each named file.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The standard input was successfully copied to all output files.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If a write to any successfully opened *file* operand fails, writes to other successfully opened *file* operands and standard output shall continue, but the exit status shall be non-zero. Otherwise, the default actions specified in Section 1.11 (on page 2221) apply.

APPLICATION USAGE

The *tee* utility is usually used in a pipeline, to make a copy of the output of some utility.

The *file* operand is technically optional, but *tee* is no more useful than *cat* when none is specified.

EXAMPLES

Save an unsorted intermediate form of the data in a pipeline:

```
... | tee unsorted | sort > sorted
```

RATIONALE

The buffering requirement means that *tee* is not allowed to use ISO C standard fully buffered or line-buffered writes. It does not mean that *tee* has to do 1-byte reads followed by 1-byte writes.

It should be noted that early versions of BSD ignore any invalid options and accept a single '–' as an alternative to **–i**. They also print a message if unable to open a file:

```
"tee: cannot access %s\n", <pathname>
```

Historical implementations ignore write errors. This is explicitly not permitted by this volume of IEEE Std 1003.1-200x.

Some historical implementations use O_APPEND when providing append mode; others use the *lseek*( ) function to seek to the end of file after opening the file without O_APPEND. This volume of IEEE Std 1003.1-200x requires functionality equivalent to using O_APPEND; see Section 1.7.1.4 (on page 2204).

FUTURE DIRECTIONS

None.

SEE ALSO

cat

CHANGE HISTORY

First released in Issue 2.

Issue 6

IEEE PASC Interpretation 1003.2 #168 is applied.

NAME

test — evaluate expression

SYNOPSIS

```
test [expression]

[ [expression] ]
```

DESCRIPTION

The *test* utility shall evaluate the *expression* and indicates the result of the evaluation by its exit status. An exit status of zero indicates that the expression evaluated as true and an exit status of 1 indicates that the expression evaluated as false.

In the second form of the utility, which uses `"[]"` rather than *test*, the application shall ensure that the square brackets are separate arguments.

OPTIONS

The *test* utility shall not recognize the `"−−"` argument in the manner specified by guideline 10 in the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

No options shall be supported.

OPERANDS

The application shall ensure that all operators and elements of primaries are presented as separate arguments to the *test* utility.

The following primaries can be used to construct *expression*:

−**b** *file*	True if *file* exists and is a block special file.
−**c** *file*	True if *file* exists and is a character special file.
−**d** *file*	True if *file* exists and is a directory.
−**e** *file*	True if *file* exists.
−**f** *file*	True if *file* exists and is a regular file.
−**g** *file*	True if *file* exists and its set-group-ID flag is set.
−**h** *file*	True if *file* exists and is a symbolic link.
−**L** *file*	True if *file* exists and is a symbolic link.
−**n** *string*	True if the length of *string* is non-zero.
−**p** *file*	True if *file* is a FIFO.
−**r** *file*	True if *file* exists and is readable. True shall indicate that permission to read from *file* will be granted, as defined in Section 1.7.1.4 (on page 2204).
−**S** *file*	True if *file* exists and is a socket.
−**s** *file*	True if *file* exists and has a size greater than zero.
−**t** *file_descriptor*	True if the file whose file descriptor number is *file_descriptor* is open and is associated with a terminal.
−**u** *file*	True if *file* exists and its set-user-ID flag is set.
−**w** *file*	True if *file* exists and is writable. True shall indicate that permission to write from *file* will be granted, as defined in Section 1.7.1.4 (on page 2204).

	−x *file*	True if *file* exists and is executable. True shall indicate that permission to execute *file* will be granted, as defined in Section 1.7.1.4 (on page 2204). If *file* is a directory, true shall indicate that permission to search *file* will be granted.
	−z *string*	True if the length of string *string* is zero.
	string	True if the string *string* is not the null string.
	s1 **=** *s2*	True if the strings *s1* and *s2* are identical.
	s1 **!=** *s2*	True if the strings *s1* and *s2* are not identical.
	n1 **−eq** *n2*	True if the integers *n1* and *n2* are algebraically equal.
	n1 **−ne** *n2*	True if the integers *n1* and *n2* are not algebraically equal.
	n1 **−gt** *n2*	True if the integer *n1* is algebraically greater than the integer *n2*.
	n1 **−ge** *n2*	True if the integer *n1* is algebraically greater than or equal to the integer *n2*.
	n1 **−lt** *n2*	True if the integer *n1* is algebraically less than the integer *n2*.
	n1 **−le** *n2*	True if the integer *n1* is algebraically less than or equal to the integer *n2*.
XSI	*expression1* **−a** *expression2*	True if both *expression1* and *expression2* are true. The **−a** binary primary is left associative. It has a higher precedence than **−o**.
XSI	*expression1* **−o** *expression2*	True if either *expression1* or *expression2* is true. The **−o** binary primary is left associative.

With the exception of the **−h** *file* primary, if a *file* argument is a symbolic link, *test* shall evaluate the expression by resolving the symbolic link and using the file referenced by the link.

These primaries can be combined with the following operators:

	! *expression*	True if *expression* is false.
XSI	**(** *expression* **)**	True if *expression* is true. The parentheses can be used to alter the normal precedence and associativity.

The primaries with two elements of the form:

```
−primary_operator primary_operand
```

are known as *unary primaries*. The primaries with three elements in either of the two forms:

```
primary_operand −primary_operator primary_operand
primary_operand primary_operator primary_operand
```

are known as *binary primaries*. Additional implementation-defined operators and *primary_operator*s may be provided by implementations. They shall be of the form −*operator* where the first character of *operator* is not a digit.

The algorithm for determining the precedence of the operators and the return value that shall be generated is based on the number of arguments presented to *test*. (However, when using the `"[...]"` form, the right-bracket final argument shall not be counted in this algorithm.)

In the following list, $1, $2, $3, and $4 represent the arguments presented to *test*:

0 arguments: Exit false (1).

1 argument:	Exit true (0) if $1 is not null; otherwise, exit false.
2 arguments:	• If $1 is `'!'`, exit true if $2 is null, false if $2 is not null. • If $1 is a unary primary, exit true if the unary test is true, false if the unary test is false. • Otherwise, produce unspecified results.
3 arguments:	• If $2 is a binary primary, perform the binary test of $1 and $3. • If $1 is `'!'`, negate the two-argument test of $2 and $3. • If $1 is `'('` and $3 is `')'`, perform the unary test of $2. • Otherwise, produce unspecified results.
4 arguments:	• If $1 is `'!'`, negate the three-argument test of $2, $3, and $4. XSI • If $1 is `'('` and $4 is `')'`, perform the two-argument test of $2 and $3. • Otherwise, the results are unspecified.
>4 arguments:	The results are unspecified. XSI On XSI-conformant systems, combinations of primaries and operators shall be evaluated using the precedence and associativity rules described previously. In addition, the string comparison binary primaries `'='` and `"!="` shall have a higher precedence than any unary primary.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *test*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	*expression* evaluated to true.
1	*expression* evaluated to false or *expression* was missing.
>1	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Scripts should be careful when dealing with user-supplied input that could be confused with primaries and operators. Unless the application writer knows all the cases that produce input to the script, invocations like:

```
test "$1" −a "$2"
```

should be written as:

```
test "$1" && test "$2"
```

to avoid problems if a user supplied values such as $1 set to '!' and $2 set to the null string. That is, in cases where maximal portability is of concern, replace:

```
test expr1 −a expr2
```

with:

```
test expr1 && test expr2
```

and replace:

```
test expr1 −o expr2
```

with:

```
test expr1 || test expr2
```

but note that, in *test*, **−a** has higher precedence than **−o** while "&&" and "||" have equal precedence in the shell.

Parentheses or braces can be used in the shell command language to effect grouping.

Parentheses must be escaped when using *sh*; for example:

```
test \( expr1 −a expr2 \) −o expr3
```

This command is not always portable outside XSI-conformant systems. The following form can be used instead:

```
( test expr1 && test expr2 ) || test expr3
```

The two commands:

```
test "$1"
test ! "$1"
```

could not be used reliably on some historical systems. Unexpected results would occur if such a *string* expression were used and $1 expanded to '!', '(', or a known unary primary. Better constructs are:

```
test -n "$1"
test -z "$1"
```

respectively.

Historical systems have also been unreliable given the common construct:

```
test "$response" = "expected string"
```

One of the following is a more reliable form:

```
test "X$response" = "Xexpected string"
test "expected string" = "$response"
```

Note that the second form assumes that *expected string* could not be confused with any unary primary. If *expected string* starts with '-', '(', '!', or even '=', the first form should be used instead. Using the preceding rules without the XSI marked extensions, any of the three comparison forms is reliable, given any input. (However, note that the strings are quoted in all cases.)

Because the string comparison binary primaries, '=' and "!=", have a higher precedence than any unary primary in the greater than 4 argument case, unexpected results can occur if arguments are not properly prepared. For example, in:

```
test -d $1 -o -d $2
```

If $1 evaluates to a possible directory name of '=', the first three arguments are considered a string comparison, which shall cause a syntax error when the second -**d** is encountered. One of the following forms prevents this; the second is preferred:

```
test \( -d "$1" \) -o \( -d "$2" \)
test -d "$1" || test -d "$2"
```

Also in the greater than 4 argument case:

```
test "$1" = "bat" -a "$2" = "ball"
```

Syntax errors occur if $1 evaluates to '(' or '!'. One of the following forms prevents this; the third is preferred:

```
test "X$1" = "Xbat" -a "X$2" = "Xball"
test "$1" = "bat" && test "$2" = "ball"
test "X$1" = "Xbat" && test "X$2" = "Xball"
```

EXAMPLES

1. Exit if there are not two or three arguments (two variations):

   ```
   if [ $# -ne 2 -a $# -ne 3 ]; then exit 1; fi
   if [ $# -lt 2 -o $# -gt 3 ]; then exit 1; fi
   ```

2. Perform a *mkdir* if a directory does not exist:

```
test ! −d tempdir && mkdir tempdir
```

3. Wait for a file to become non-readable:

```
while test −r thefile
do
    sleep 30
done
echo '"thefile" is no longer readable'
```

4. Perform a command if the argument is one of three strings (two variations):

```
if [ "$1" = "pear" ] || [ "$1" = "grape" ] || [ "$1" = "apple" ]
then
    command
fi

case "$1" in
    pear|grape|apple) command ;;
esac
```

RATIONALE

The KornShell-derived conditional command (double bracket **[[ ]]**) was removed from the shell command language description in an early proposal. Objections were raised that the real problem is misuse of the *test* command (**[**), and putting it into the shell is the wrong way to fix the problem. Instead, proper documentation and a new shell reserved word (**!**) are sufficient.

Tests that require multiple *test* operations can be done at the shell level using individual invocations of the *test* command and shell logicals, rather than using the error-prone −**o** flag of *test*.

XSI-conformant systems support more than four arguments.

XSI-conformant systems support the combining of primaries with the following constructs:

expression1 −**a** *expression2*
True if both *expression1* and *expression2* are true.

expression1 −**o** *expression2*
True if at least one of *expression1* and *expression2* are true.

(*expression* **)**
True if *expression* is true.

In evaluating these more complex combined expressions, the following precedence rules are used:

- The unary primaries have higher precedence than the algebraic binary primaries.
- The unary primaries have lower precedence than the string binary primaries.
- The unary and binary primaries have higher precedence than the unary *string* primary.
- The **!** operator has higher precedence than the −**a** operator, and the −**a** operator has higher precedence than the −**o** operator.
- The −**a** and −**o** operators are left associative.
- The parentheses can be used to alter the normal precedence and associativity.

The BSD and System V versions of −**f** are not the same. The BSD definition was:

−**f** *file*	True if *file* exists and is not a directory.

The SVID version (true if the file exists and is a regular file) was chosen for this volume of IEEE Std 1003.1-200x because its use is consistent with the −**b**, −**c**, −**d**, and −**p** operands (*file* exists and is a specific file type).

The −**e** primary, possessing similar functionality to that provided by the C shell, was added because it provides the only way for a shell script to find out if a file exists without trying to open the file. Since implementations are allowed to add additional file types, a portable script cannot use:

```
test −b foo −o −c foo −o −d foo −o −f foo −o −p foo
```

to find out if **foo** is an existing file.) On historical BSD systems, the existence of a file could be determined by:

```
test −f foo −o −d foo
```

but there was no easy way to determine that an existing file was a regular file. An early proposal used the KornShell −**a** primary (with the same meaning), but this was changed to −**e** because there were concerns about the high probability of humans confusing the −**a** primary with the −**a** binary operator.

The following options were not included in this volume of IEEE Std 1003.1-200x, although they are provided by some implementations. These operands should not be used by new implementations for other purposes:

−**k** *file*	True if *file* exists and its sticky bit is set.
−**C** *file*	True if *file* is a contiguous file.
−**V** *file*	True if *file* is a version file.

The following option was not included because it was undocumented in most implementations, has been removed from some implementations (including System V), and the functionality is provided by the shell (see Section 2.6.2 (on page 2239).

−**l** *string*	The length of the string *string*.

The −**b**, −**c**, −**g**, −**p**, −**u**, and −**x** operands are derived from the SVID; historical BSD does not provide them. The −**k** operand is derived from System V; historical BSD does not provide it.

On historical BSD systems, *test* −**w** *directory* always returned false because *test* tried to open the directory for writing, which always fails.

Some additional primaries newly invented or from the KornShell appeared in an early proposal as part of the conditional command (**[[ ]]**): *s1* **>** *s2*, *s1* **<** *s2*, *str* **=** *pattern*, *str* **!=** *pattern*, *f1* −**nt** *f2*, *f1* −**ot** *f2*, and *f1* −**ef** *f2*. They were not carried forward into the *test* utility when the conditional command was removed from the shell because they have not been included in the *test* utility built into historical implementations of the *sh* utility.

The −**t** *file_descriptor* primary is shown with a mandatory argument because the grammar is ambiguous if it can be omitted. Historical implementations have allowed it to be omitted, providing a default of 1.

FUTURE DIRECTIONS

None.

SEE ALSO

find

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

The −**h** operand is added for symbolic links, and access permission requirements are clarified for the −**r**, −**w**, and −**x** operands to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The −**L** and −**S** operands are added for symbolic links and sockets.

NAME

time — time a simple command

SYNOPSIS

UP
```
time [−p] utility [argument...]
```

DESCRIPTION

The *time* utility shall invoke the utility named by the *utility* operand with arguments supplied as the *argument* operands and write a message to standard error that lists timing statistics for the utility. The message shall include the following information:

- The elapsed (real) time between invocation of *utility* and its termination.
- The User CPU time, equivalent to the sum of the *tms_utime* and *tms_cutime* fields returned by the *times*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x for the process in which *utility* is executed.
- The System CPU time, equivalent to the sum of the *tms_stime* and *tms_cstime* fields returned by the *times*( ) function for the process in which *utility* is executed.

The precision of the timing shall be no less than the granularity defined for the size of the clock tick unit on the system, but the results shall be reported in terms of standard time units (for example, 0.02 seconds, 00:00:00.02, 1m33.75s, 365.21 seconds), not numbers of clock ticks.

When *time* is used as part of a pipeline, the times reported are unspecified, except when it is the sole command within a grouping command (see Section 2.9.4.1 (on page 2253)) in that pipeline. For example, the commands on the left are unspecified; those on the right report on utilities **a** and **c**, respectively:

```
time a | b | c     { time a } | b | c
a | b | time c     a | b | (time c)
```

OPTIONS

The *time* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−p	Write the timing output to standard error in the format shown in the STDERR section.

OPERANDS

The following operands shall be supported:

utility	The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 (on page 2266), the results are undefined.
argument	Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *time*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic and informative messages written to standard error.
LC_NUMERIC	Determine the locale for numeric formatting.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the search path that shall be used to locate the utility to be invoked; see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used to write the timing statistics. If −**p** is specified, the following format shall be used in the POSIX locale:

```
"real %f\nuser %f\nsys %f\n", <real seconds>, <user seconds>,
    <system seconds>
```

where each floating-point number shall be expressed in seconds. The precision used may be less than the default six digits of `%f`, but shall be sufficiently precise to accommodate the size of the clock tick on the system (for example, if there were 60 clock ticks per second, at least two digits shall follow the radix character). The number of digits following the radix character shall be no less than one, even if this always results in a trailing zero. The implementation may append white space and additional information following the format shown here.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If the *utility* utility is invoked, the exit status of *time* shall be the exit status of *utility*; otherwise, the *time* utility shall exit with one of the following values:

1-125 An error occurred in the *time* utility.

126	The utility specified by *utility* was found but could not be invoked.
127	The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *command*, *env*, *nice*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

It is frequently desirable to apply *time* to pipelines or lists of commands. This can be done by placing pipelines and command lists in a single file; this file can then be invoked as a utility, and the *time* applies to everything in the file.

Alternatively, the following command can be used to apply *time* to a complex command:

```
time sh −c 'complex-command-line'
```

RATIONALE

When the *time* utility was originally proposed to be included in the earlier version of IEEE Std 1003.1, questions were raised about its suitability for inclusion on the grounds that it was not useful for portable applications, specifically:

- The underlying CPU definitions from the System Interfaces volume of IEEE Std 1003.1-200x are vague, so the numeric output could not be compared accurately between systems or even between invocations.
- The creation of portable benchmark programs was outside the scope this volume of IEEE Std 1003.1-200x.

However, *time* does fit in the scope of user portability. Human judgement can be applied to the analysis of the output, and it could be very useful in hands-on debugging of applications or in providing subjective measures of system performance. Hence it has been included in this volume of IEEE Std 1003.1-200x.

The default output format has been left unspecified because historical implementations differ greatly in their style of depicting this numeric output. The **−p** option was invented to provide scripts a common means of obtaining this information.

In the KornShell, *time* is a shell reserved word that can be used to time an entire pipeline, rather than just a simple command. The POSIX definition has been worded to allow this implementation. Consideration was given to invalidating this approach because of the historical model from the C shell and System V shell. However, since the System V *time* utility historically has not produced accurate results in pipeline timing (because the constituent processes are not all owned by the same parent process, as allowed by POSIX), it did not seem worthwhile to break historical KornShell usage.

The term *utility* is used, rather than *command*, to highlight the fact that shell compound commands, pipelines, special built-ins, and so on, cannot be used directly. However, *utility* includes user application programs and shell scripts, not just the standard utilities.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the System Interfaces volume of IEEE Std 1003.1-200x, *times*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

NAME

touch — change file access and modification times

SYNOPSIS

```
touch [−acm][ −r ref_file| −t time] file...
```

DESCRIPTION

The *touch* utility shall change the modification times, access times, or both of files. The modification time shall be equivalent to the value of the *st_mtime* member of the **stat** structure for a file, as described in the System Interfaces volume of IEEE Std 1003.1-200x; the access time shall be equivalent to the value of *st_atime*.

The time used can be specified by the **−t** *time* option-argument, the corresponding time fields of the file referenced by the **−r** *ref_file* option-argument, or the *date_time* operand, as specified in the following sections. If none of these are specified, *touch* shall use the current time (the value returned by the equivalent of the *time*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x).

For each *file* operand, *touch* shall perform actions equivalent to the following functions defined in the System Interfaces volume of IEEE Std 1003.1-200x:

1. If *file* does not exist, a *creat*( ) function call is made with the *file* operand used as the *path* argument and the value of the bitwise-inclusive OR of S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH, and S_IWOTH used as the *mode* argument.
2. The *utime*( ) function is called with the following arguments:
   a. The *file* operand is used as the *path* argument.
   b. The **utimbuf** structure members *actime* and *modtime* are determined as described in the OPTIONS section.

OPTIONS

The *touch* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Change the access time of *file*. Do not change the modification time unless **−m** is also specified.
−c	Do not create a specified *file* if it does not exist. Do not write any diagnostic messages concerning this condition.
−m	Change the modification time of *file*. Do not change the access time unless **−a** is also specified.
−r *ref_file*	Use the corresponding time of the file named by the pathname *ref_file* instead of the current time.
−t *time*	Use the specified *time* instead of the current time. The option-argument shall be a decimal number of the form: **[[***CC***]***YY***]***MMDDhhmm***[.***SS***]** where each two digits represents the following: *MM* The month of the year [01,12]. *DD* The day of the month [01,31].

hh	The hour of the day [00,23].
mm	The minute of the hour [00,59].
CC	The first two digits of the year (the century).
YY	The second two digits of the year.
SS	The second of the minute [00,60].

Both *CC* and *YY* shall be optional. If neither is given, the current year shall be assumed. If *YY* is specified, but *CC* is not, *CC* shall be derived as follows:

If *YY* is:	***CC* becomes:**
[69,99]	19
[00,68]	20

Note: It is expected that in a future version of IEEE Std 1003.1-200x the default century inferred from a 2-digit year will change. (This would apply to all commands accepting a 2-digit year as input.)

The resulting time shall be affected by the value of the *TZ* environment variable. If the resulting time value precedes the Epoch, *touch* shall exit immediately with an error status. The range of valid times past the Epoch is implementation-defined, but it shall extend to at least the time 0 hours, 0 minutes, 0 seconds, January 1, 2038, Coordinated Universal Time. Some implementations may not be able to represent dates beyond the January 18, 2038, because they use **signed int** as a time holder.

The range for *SS* is [00,60] rather than [00,59] because of leap seconds. If *SS* is 60, and the resulting time, as affected by the *TZ* environment variable, does not refer to a leap second, the resulting time shall be one second after a time where *SS* is 59. If *SS* is not given a value, it is assumed to be zero.

If neither the −**a** nor −**m** options were specified, *touch* shall behave as if both the −**a** and −**m** options were specified.

OPERANDS

The following operands shall be supported:

file	A pathname of a file whose times shall be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *touch*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in

arguments).

LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
TZ	Determine the timezone to be used for interpreting the *time* option-argument. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The utility executed successfully and all requested changes were made.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The interpretation of time is taken to be *seconds since the Epoch* (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 4.14, Seconds Since the Epoch). It should be noted that implementations conforming to the System Interfaces volume of IEEE Std 1003.1-200x do not take leap seconds into account when computing seconds since the Epoch. When *SS*=60 is used, the resulting time always refers to 1 plus *seconds since the Epoch* for a time when *SS*=59.

Although the **–t** *time* option-argument specifies values in 1969, the access time and modification time fields are defined in terms of seconds since the Epoch (00:00:00 on 1 January 1970 UTC). Therefore, depending on the value of *TZ* when *touch* is run, there is never more than a few valid hours in 1969 and there need not be any valid times in 1969.

One ambiguous situation occurs if **–t** *time* is not specified, **–r** *ref_file* is not specified, and the first operand is an eight or ten-digit decimal number. A portable script can avoid this problem by using:

```
touch -- file
```

or:

```
touch ./file
```

in this case.

EXAMPLES

None.

RATIONALE

The functionality of *touch* is described almost entirely through references to functions in the System Interfaces volume of IEEE Std 1003.1-200x. In this way, there is no duplication of effort required for describing such side effects as the relationship of user IDs to the user database, permissions, and so on.

There are some significant differences between the *touch* utility in this volume of IEEE Std 1003.1-200x and those in System V and BSD systems. They are upward-compatible for historical applications from both implementations:

1. In System V, an ambiguity exists when a pathname that is a decimal number leads the operands; it is treated as a time value. In BSD, no *time* value is allowed; files may only be *touch*ed to the current time. The −**t** *time* construct solves these problems for future conforming applications (note that the −**t** option is not historical practice).
2. The inclusion of the century digits, *CC*, is also new. Note that a ten-digit *time* value is treated as if *YY*, and not *CC*, were specified. The caveat about the range of dates following the Epoch was included as recognition that some implementations are not able to represent dates beyond 18 January 2038 because they use **signed int** as a time holder.

The −**r** option was added because several comments requested this capability. This option was named −**f** in an early proposal, but was changed because the −**f** option is used in the BSD version of *touch* with a different meaning.

At least one historical implementation of *touch* incremented the exit code if −**c** was specified and the file did not exist. This volume of IEEE Std 1003.1-200x requires exit status zero if no errors occur.

FUTURE DIRECTIONS

Applications should use the −**r** or −**t** options.

SEE ALSO

date, the System Interfaces volume of IEEE Std 1003.1-200x, *creat*( ), *time*( ), **<sys/stat.h>**

CHANGE HISTORY

First released in Issue 2.

Issue 6

The obsolescent *date_time* operand is removed.

The Open Group Corrigendum U027/1 is applied. This extends the range of valid time past the Epoch to at least the time 0 hours, 0 minutes, 0 seconds, January 1, 2038, Coordinated Universal Time. This is a new requirement on POSIX implementations.

The range for double leap seconds is changed from [00,61] to [00,60] to align with the ISO/IEC 9899:1999 standard.

NAME

tput — change terminal characteristics

SYNOPSIS

UP
```
tput [−T type] operand...
```

DESCRIPTION

The *tput* utility shall display terminal-dependent information. The manner in which this information is retrieved is unspecified. The information displayed shall clear the terminal screen, initialize the user's terminal, or reset the user's terminal, depending on the operand given. The exact consequences of displaying this information are unspecified.

OPTIONS

The *tput* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−T *type*	Indicate the type of terminal. If this option is not supplied and the *TERM* variable is unset or null, an unspecified default terminal type shall be used. The setting of *type* shall take precedence over the value in *TERM*.

OPERANDS

The following strings shall be supported as operands by the implementation in the POSIX locale:

clear	Display the clear-screen sequence.
init	Display the sequence that initializes the user's terminal in an implementation-defined manner.
reset	Display the sequence that resets the user's terminal in an implementation-defined manner.

If a terminal does not support any of the operations described by these operands, this shall not be considered an error condition.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tput*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI	*NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
	TERM	Determine the terminal type. If this variable is unset or null, and if the −**T** option is not specified, an unspecified default terminal type shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If standard output is a terminal device, it may be used for writing the appropriate sequence to clear the screen or reset or initialize the terminal. If standard output is not a terminal device, undefined results occur.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The requested string was written successfully.
1	Unspecified.
2	Usage error.
3	No information is available about the specified terminal type.
4	The specified operand is invalid.
>4	An error occurred.

CONSEQUENCES OF ERRORS

If one of the operands is not available for the terminal, *tput* continues processing the remaining operands.

APPLICATION USAGE

The difference between resetting and initializing a terminal is left unspecified, as they vary greatly based on hardware types. In general, resetting is a more severe action.

Some terminals use control characters to perform the stated functions, and on such terminals it might make sense to use *tput* to store the initialization strings in a file or environment variable for later use. However, because other terminals might rely on system calls to do this work, the standard output cannot be used in a portable manner, such as the following non-portable constructs:

```
ClearVar='tput clear'
tput reset | mailx −s "Wake Up" ddg
```

EXAMPLES

1. Initialize the terminal according to the type of terminal in the environmental variable *TERM*. This command can be included in a **.profile** file.

   ```
   tput init
   ```

2. Reset a 450 terminal.

```
tput −T 450 reset
```

RATIONALE

The list of operands was reduced to a minimum for the following reasons:

- The only features chosen were those that were likely to be used by human users interacting with a terminal.
- Specifying the full *terminfo* set was not considered desirable, but the standard developers did not want to select among operands.
- This volume of IEEE Std 1003.1-200x does not attempt to provide applications with sophisticated terminal handling capabilities, as that falls outside of its assigned scope and intersects with the responsibilities of other standards bodies.

The difference between resetting and initializing a terminal is left unspecified as this varies greatly based on hardware types. In general, resetting is a more severe action.

The exit status of 1 is historically reserved for finding out if a Boolean operand is not set. Although the operands were reduced to a minimum, the exit status of 1 should still be reserved for the Boolean operands, for those sites that wish to support them.

FUTURE DIRECTIONS

None.

SEE ALSO

stty, *tabs*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

NAME

tr — translate characters

SYNOPSIS

```
tr [−c | −C][−s] string1 string2

tr −s [−c | −C] string1

tr −d [−c | −C] string1

tr −ds [−c | −C] string1 string2
```

DESCRIPTION

The *tr* utility shall copy the standard input to the standard output with substitution or deletion of selected characters. The options specified and the *string1* and *string2* operands shall control translations that occur while copying characters and single-character collating elements.

OPTIONS

The *tr* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	Complement the set of values specified by *string1*. See the EXTENDED DESCRIPTION section.
−C	Complement the set of characters specified by *string1*. See the EXTENDED DESCRIPTION section.
−d	Delete all occurrences of input characters that are specified by *string1*.
−s	Replace instances of repeated characters with a single character, as described in the EXTENDED DESCRIPTION section.

OPERANDS

The following operands shall be supported:

string1, *string2*

Translation control strings. Each string shall represent a set of characters to be converted into an array of characters used for the translation. For a detailed description of how the strings are interpreted, see the EXTENDED DESCRIPTION section.

STDIN

The standard input can be any type of file.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tr*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of range expressions and equivalence classes.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments) and the behavior of character classes.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *tr* output shall be identical to the input, with the exception of the specified transformations.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The operands *string1* and *string2* (if specified) define two arrays of characters. The constructs in the following list can be used to specify characters or single-character collating elements. If any of the constructs result in multi-character collating elements, *tr* shall exclude, without a diagnostic, those multi-character elements from the resulting array.

character	Any character not described by one of the conventions below shall represent itself.
octal	Octal sequences can be used to represent characters with specific coded values. An octal sequence shall consist of a backslash followed by the longest sequence of one, two, or three-octal-digit characters (01234567). The sequence shall cause the value whose encoding is represented by the one, two, or three-digit octal integer to be placed into the array. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-defined. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading `'\'` for each byte.
character	The backslash-escape sequences in the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`) shall be supported. The results of using any other character, other than an octal digit, following the backslash are unspecified.
c−*c*	In the POSIX locale, this construct shall represent the range of collating elements between the range endpoints (as long as neither endpoint is an octal sequence of the form *octal*), inclusive, as defined by the collation sequence. The characters or collating elements in the range shall be placed in the array in ascending collation sequence. If the second endpoint precedes the starting endpoint in the collation sequence, it is unspecified whether the range of collating elements is empty, or this construct is treated as invalid. In locales other than the POSIX locale, this construct has unspecified behavior. If either or both of the range endpoints are octal sequences of the form *octal*, this shall represent the range of specific coded values between the two range endpoints, inclusive.

[:*class*:] Represents all characters belonging to the defined character class, as defined by the current setting of the *LC_CTYPE* locale category. The following character class names shall be accepted when specified in *string1*:

alnum **blank** **digit** **lower** **punct** **upper**
alpha **cntrl** **graph** **print** **space** **xdigit**

XSI In addition, character class expressions of the form [:*name*:] shall be recognized in those locales where the *name* keyword has been given a **charclass** definition in the *LC_CTYPE* category.

When both the **–d** and **–s** options are specified, any of the character class names shall be accepted in *string2*. Otherwise, only character class names **lower** or **upper** are valid in *string2* and then only if the corresponding character class (**upper** and **lower**, respectively) is specified in the same relative position in *string1*. Such a specification shall be interpreted as a request for case conversion. When [: *lower*:] appears in *string1* and [:*upper*:] appears in *string2*, the arrays shall contain the characters from the **toupper** mapping in the *LC_CTYPE* category of the current locale. When [:*upper*:] appears in *string1* and [:*lower*:] appears in *string2*, the arrays shall contain the characters from the **tolower** mapping in the *LC_CTYPE* category of the current locale. The first character from each mapping pair shall be in the array for *string1* and the second character from each mapping pair shall be in the array for *string2* in the same relative position.

Except for case conversion, the characters specified by a character class expression shall be placed in the array in an unspecified order.

If the name specified for *class* does not define a valid character class in the current locale, the behavior is undefined.

[=*equiv*=] Represents all characters or collating elements belonging to the same equivalence class as *equiv*, as defined by the current setting of the *LC_COLLATE* locale category. An equivalence class expression shall be allowed only in *string1*, or in *string2* when it is being used by the combined **–d** and **–s** options. The characters belonging to the equivalence class shall be placed in the array in an unspecified order.

[*x***n*] Represents *n* repeated occurrences of the character *x*. Because this expression is used to map multiple characters to one, it is only valid when it occurs in *string2*. If *n* is omitted or is zero, it shall be interpreted as large enough to extend the *string2*-based sequence to the length of the *string1*-based sequence. If *n* has a leading zero, it shall be interpreted as an octal value. Otherwise, it shall be interpreted as a decimal value.

When the **–d** option is not specified:

- Each input character found in the array specified by *string1* shall be replaced by the character in the same relative position in the array specified by *string2*. When the array specified by *string2* is shorter that the one specified by *string1*, the results are unspecified.
- If the **–C** option is specified, the complements of the characters specified by *string1* (the set of all characters in the current character set, as defined by the current setting of *LC_CTYPE*, except for those actually specified in the *string1* operand) shall be placed in the array in ascending collation sequence, as defined by the current setting of *LC_COLLATE*.
- If the **–c** option is specified, the complement of the values specified by *string1* shall be placed in the array in ascending order by binary value.

- Because the order in which characters specified by character class expressions or equivalence class expressions is undefined, such expressions should only be used if the intent is to map several characters into one. An exception is case conversion, as described previously.

When the −**d** option is specified:

- Input characters found in the array specified by *string1* shall be deleted.
- When the −**C** option is specified with −**d**, all characters except those specified by *string1* shall be deleted. The contents of *string2* are ignored, unless the −**s** option is also specified.
- When the −**c** option is specified with −**d**, all values except those specified by *string1* shall be deleted. The contents of *string2* shall be ignored, unless the −**s** option is also specified.
- The same string cannot be used for both the −**d** and the −**s** option; when both options are specified, both *string1* (used for deletion) and *string2* (used for squeezing) shall be required.

When the −**s** option is specified, after any deletions or translations have taken place, repeated sequences of the same character shall be replaced by one occurrence of the same character, if the character is found in the array specified by the last operand. If the last operand contains a character class, such as the following example:

```
tr −s '[:space:]'
```

the last operand's array shall contain all of the characters in that character class. However, in a case conversion, as described previously, such as:

```
tr −s '[:upper:]' '[:lower:]'
```

the last operand's array shall contain only those characters defined as the second characters in each of the **toupper** or **tolower** character pairs, as appropriate.

An empty string used for *string1* or *string2* produces undefined results.

EXIT STATUS

The following exit values shall be returned:

0	All input was processed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If necessary, *string1* and *string2* can be quoted to avoid pattern matching by the shell.

If an ordinary digit (representing itself) is to follow an octal sequence, the octal sequence must use the full three digits to avoid ambiguity.

When *string2* is shorter than *string1*, a difference results between historical System V and BSD systems. A BSD system pads *string2* with the last character found in *string2*. Thus, it is possible to do the following:

```
tr 0123456789 d
```

which would translate all digits to the letter `'d'`. Since this area is specifically unspecified in this volume of IEEE Std 1003.1-200x, both the BSD and System V behaviors are allowed, but a conforming application cannot rely on the BSD behavior. It would have to code the example in the following way:

```
tr 0123456789 '[d*]'
```

It should be noted that, despite similarities in appearance, the string operands used by *tr* are not regular expressions.

Unlike some historical implementations, this definition of the *tr* utility correctly processes NUL characters in its input stream. NUL characters can be stripped by using:

```
tr −d '\000'
```

EXAMPLES

1. The following example creates a list of all words in **file1** one per line in **file2**, where a word is taken to be a maximal string of letters.

   ```
   tr −cs "[:alpha:]" "[\n*]" <file1 >file2
   ```

2. The next example translates all lowercase characters in **file1** to uppercase and writes the results to standard output.

   ```
   tr "[:lower:]" "[:upper:]" <file1
   ```

3. This example uses an equivalence class to identify accented variants of the base character `'e'` in **file1**, which are stripped of diacritical marks and written to **file2**.

   ```
   tr "[=e=]" e <file1 >file2
   ```

RATIONALE

In some early proposals, an explicit option −**n** was added to disable the historical behavior of stripping NUL characters from the input. It was considered that automatically stripping NUL characters from the input was not correct functionality. However, the removal of −**n** in a later proposal does not remove the requirement that *tr* correctly process NUL characters in its input stream. NUL characters can be stripped by using *tr* −**d** '\000'.

Historical implementations of *tr* differ widely in syntax and behavior. For example, the BSD version has not needed the bracket characters for the repetition sequence. The POSIX Shell and Utilities *tr* syntax is based more closely on the System V and XPG3 model while attempting to accommodate historical BSD implementations. In the case of the short *string2* padding, the decision was to unspecify the behavior and preserve System V and XPG3 scripts, which might find difficulty with the BSD method. The assumption was made that BSD users of *tr* have to make accommodations to meet the POSIX Shell and Utilities syntax. Since it is possible to use the repetition sequence to duplicate the desired behavior, whereas there is no simple way to achieve the System V method, this was the correct, if not desirable, approach.

The use of octal values to specify control characters, while having historical precedents, is not portable. The introduction of escape sequences for control characters should provide the necessary portability. It is recognized that this may cause some historical scripts to break.

An early proposal included support for multi-character collating elements. It was pointed out that, while *tr* does employ some syntactical elements from REs, the aim of *tr* is quite different; ranges, for example, do not have a similar meaning (‘‘any of the chars in the range matches’’, *versus* ‘‘translate each character in the range to the output counterpart’’). As a result, the previously included support for multi-character collating elements has been removed. What remains are ranges in current collation order (to support, for example, accented characters), character classes, and equivalence classes.

In XPG3 the [:*class*:] and [=*equiv*=] conventions are shown with double brackets, as in RE syntax. However, *tr* does not implement RE principles; it just borrows part of the syntax. Consequently, [:*class*:] and [=*equiv*=] should be regarded as syntactical elements on a par with [*x***n*], which is not an RE bracket expression.

The standard developers will consider changes to *tr* that allow it to translate characters between different character encodings, or they will consider providing a new utility to accomplish this.

On historical System V systems, a range expression requires enclosing square-brackets, such as:

```
tr '[a-z]' '[A-Z]'
```

However, BSD-based systems did not require the brackets, and this convention is used by POSIX Shell and Utilities to avoid breaking large numbers of BSD scripts:

```
tr a-z A-Z
```

The preceding System V script will continue to work because the brackets, treated as regular characters, are translated to themselves. However, any System V script that relied on *a-z* representing the three characters `'−,'` and `'z'` have to be rewritten as *az*−.

A prior version of IEEE Std 1003.1-200x had a −**c** option that behaved similarly to the −**C** option, but did not supply functionality equivalent to the −**c** option specified in IEEE Std 1003.1-200x. This meant that historical practice of being able to specify *tr* −**d**\200−\377 (which would delete all bytes with the top bit set) would have no effect because, in the C locale, bytes with the values octal 200 to octal 377 are not characters.

The earlier version also said that octal sequences referred to collating elements and could be placed adjacent to each other to specify multi-byte characters. However, it was noted that this caused ambiguities because *tr* would not be able to tell whether adjacent octal sequences were intending to specify multi-byte characters or multiple single byte characters. IEEE Std 1003.1-200x specifies that octal sequences always refer to single byte binary values.

FUTURE DIRECTIONS

None.

SEE ALSO

sed

CHANGE HISTORY

First released in Issue 2.

Issue 6

The −**C** operand is added, and the description of the −**c** operand is changed to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

true — return true value

SYNOPSIS

```
true
```

DESCRIPTION

The *true* utility shall return with exit code zero.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Default.

CONSEQUENCES OF ERRORS

None.

APPLICATION USAGE

This utility is typically used in shell scripts, as shown in the EXAMPLES section. The special built-in utility `:` is sometimes more efficient than *true*.

EXAMPLES

This command is executed forever:

```
while true
do
    command
done
```

RATIONALE

The *true* utility has been retained in this volume of IEEE Std 1003.1-200x, even though the shell special built-in **:** provides similar functionality, because *true* is widely used in historical scripts and is less cryptic to novice script readers.

FUTURE DIRECTIONS

None.

SEE ALSO

false, Section 2.9 (on page 2248)

CHANGE HISTORY

First released in Issue 2.

NAME

tsort — topological sort

SYNOPSIS

XSI

```
tsort [file]
```

DESCRIPTION

The *tsort* utility shall write to standard output a totally ordered list of items consistent with a partial ordering of items contained in the input.

The application shall ensure that the input consists of pairs of items (non-empty strings) separated by <blank>s. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

file	A pathname of a text file to order. If no *file* operand is given, the standard input shall be used.

STDIN

The standard input shall be a text file that is used if no *file* operand is given.

INPUT FILES

The input file named by the *file* operand is a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tsort*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be a text file consisting of the order list produced from the partially ordered input.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *LC_COLLATE* variable need not affect the actions of *tsort*. The output ordering is not lexicographic, but depends on the pairs of items given as input.

EXAMPLES

The command:

```
tsort <<EOF
a b c c d e
g g
f g e f
h h
EOF
```

produces the output:

```
a
b
c
d
e
f
g
h
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

tty — return user's terminal name

SYNOPSIS

```
tty
```

DESCRIPTION

The *tty* utility shall write to the standard output the name of the terminal that is open as standard input. The name that is used shall be equivalent to the string that would be returned by the *ttyname*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x.

OPTIONS

The *tty* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

OPERANDS

None.

STDIN

While no input is read from standard input, standard input shall be examined to determine whether or not it is a terminal, and, if so, to determine the name of the terminal.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *tty*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If standard input is a terminal device, a pathname of the terminal as specified by the *ttyname*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x shall be written in the following format:

```
"%s\n", <terminal name>
```

Otherwise, a message shall be written indicating that standard input is not connected to a terminal. In the POSIX locale, the *tty* utility shall use the format:

```
"not a tty\n"
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Standard input is a terminal.
1	Standard input is not a terminal.
>1	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility checks the status of the file open as standard input against that of a implementation-defined set of files. It is possible that no match can be found, or that the match found need not be the same file as that which was opened for standard input (although they are the same device).

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *isatty*( ), *ttyname*( )

CHANGE HISTORY

First released in Issue 2.

Issue 5

The SYNOPSIS is changed to indicate two forms of the command, with the second form marked as obsolete. This is a clarification and does not change the functionality published in previous issues.

Issue 6

The obsolescent –**s** option is removed.

NAME

type — write a description of command type

SYNOPSIS

XSI
```
type name...
```

DESCRIPTION

The *type* utility shall indicate how each argument would be interpreted if used as a command name.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

name	A name to be interpreted.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *type*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the location of *name*, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output of *type* contains information about each operand in an unspecified format. The information provided typically identifies the operand as a shell built-in, function, alias, or keyword, and where applicable, may display the operand's pathname.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *type* must be aware of the contents of the current shell execution environment (such as the lists of commands, functions, and built-ins processed by *hash*), it is always provided as a shell regular built-in. If it is called in a separate utility execution environment, such as one of the following:

```
nohup type writer
find . −type f | xargs type
```

it might not produce accurate results.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

command

CHANGE HISTORY

First released in Issue 2.

NAME

ulimit — set or report file size limit

SYNOPSIS

XSI `ulimit [−f][blocks]`

DESCRIPTION

The *ulimit* utility shall set or report the file-size writing limit imposed on files written by the shell and its child processes (files of any size may be read). Only a process with appropriate privileges can increase the limit.

OPTIONS

The *ulimit* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−f	Set (or report, if no *blocks* operand is present), the file size limit in blocks. The **−f** option shall also be the default case.

OPERANDS

The following operand shall be supported:

blocks	The number of 512-byte blocks to use as the new file size limit.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ulimit*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be used when no *blocks* operand is present. If the current number of blocks is limited, the number of blocks in the current limit shall be written in the following format:

```
"%d\n", <number of 512-byte blocks>
```

If there is no current limit on the number of blocks, in the POSIX locale the following format shall be used:

```
"unlimited\n"
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	A request for a higher limit was rejected or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *ulimit* affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in separate utility execution environment, such as one of the following:

```
nohup ulimit −f 10000
env ulimit 10000
```

it does not affect the file size limit of the caller's environment.

Once a limit has been decreased by a process, it cannot be increased (unless appropriate privileges are involved), even back to the original system limit.

EXAMPLES

Set the file size limit to 51 200 bytes:

```
ulimit −f 100
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *ulimit*( )

CHANGE HISTORY

First released in Issue 2.

NAME

umask — get or set the file mode creation mask

SYNOPSIS

```
umask [−S][mask]
```

DESCRIPTION

The *umask* utility shall set the file mode creation mask of the current shell execution environment (see Section 2.12 (on page 2263)) to the value specified by the *mask* operand. This mask shall affect the initial value of the file permission bits of subsequently created files. If *umask* is called in a subshell or separate utility execution environment, such as one of the following:

```
(umask 002)
nohup umask ...
find . −exec umask ... \;
```

it shall not affect the file mode creation mask of the caller's environment.

If the *mask* operand is not specified, the *umask* utility shall write to standard output the value of the invoking process's file mode creation mask.

OPTIONS

The *umask* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−S Produce symbolic output.

The default output style is unspecified, but shall be recognized on a subsequent invocation of *umask* on the same system as a *mask* operand to restore the previous file mode creation mask.

OPERANDS

The following operand shall be supported:

mask A string specifying the new file mode creation mask. The string is treated in the same way as the *mode* operand described in the EXTENDED DESCRIPTION section for *chmod*. |

For a *symbolic_mode* value, the new value of the file mode creation mask shall be the logical complement of the file permission bits portion of the file mode specified by the *symbolic_mode* string.

In a *symbolic_mode* value, the permissions *op* characters `'+'` and `'−'` shall be interpreted relative to the current file mode creation mask; `'+'` shall cause the bits for the indicated permissions to be cleared in the mask; `'−'` shall cause the bits for the indicated permissions to be set in the mask.

The interpretation of *mode* values that specify file mode bits other than the file permission bits is unspecified.

In the octal integer form of *mode*, the specified bits are set in the file mode creation mask.

The file mode creation mask shall be set to the resulting numeric value.

The default output of a prior invocation of *umask* on the same system with no operand also shall be recognized as a *mask* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *umask*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the *mask* operand is not specified, the *umask* utility shall write a message to standard output that can later be used as a *umask mask* operand.

If −**S** is specified, the message shall be in the following format:

```
"u=%s,g=%s,o=%s\n", <owner permissions>, <group permissions>,
    <other permissions>
```

where the three values shall be combinations of letters from the set {*r*, *w*, *x*}; the presence of a letter shall indicate that the corresponding bit is clear in the file mode creation mask.

If a *mask* operand is specified, there shall be no output written to standard output.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The file mode creation mask was successfully changed, or no *mask* operand was supplied.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *umask* affects the current shell execution environment, it is generally provided as a shell regular built-in.

In contrast to the negative permission logic provided by the file mode creation mask and the octal number form of the *mask* argument, the symbolic form of the *mask* argument specifies those permissions that are left alone.

EXAMPLES

Either of the commands:

```
umask a=rx,ug+w
```

```
umask 002
```

sets the mode mask so that subsequently created files have their S_IWOTH bit cleared.

After setting the mode mask with either of the above commands, the *umask* command can be used to write out the current value of the mode mask:

```
$ umask
0002
```

(The output format is unspecified, but historical implementations use the octal integer mode format.)

```
$ umask -S
u=rwx,g=rwx,o=rx
```

Either of these outputs can be used as the mask operand to a subsequent invocation of the *umask* utility.

Assuming the mode mask is set as above, the command:

```
umask g-w
```

sets the mode mask so that subsequently created files have their S_IWGRP and S_IWOTH bits cleared.

The command:

```
umask -- -w
```

sets the mode mask so that subsequently created files have all their write bits cleared. Note that *mask* operands **−r**, **−w**, **−x** or anything beginning with a hyphen, must be preceded by "−−" to keep it from being interpreted as an option.

RATIONALE

Since *umask* affects the current shell execution environment, it is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(umask 002)
nohup umask ...
find . -exec umask ... \;
```

it does not affect the file mode creation mask of the environment of the caller.

The description of the historical utility was modified to allow it to use the symbolic modes of *chmod*. The **−s** option used in early proposals was changed to **−S** because **−s** could be confused with a *symbolic_mode* form of mask referring to the S_ISUID and S_ISGID bits. |

The default output style is implementation-defined to permit implementors to provide migration to the new symbolic style at the time most appropriate to their users. An **–o** flag to force octal mode output was omitted because the octal mode may not be sufficient to specify all of the information that may be present in the file mode creation mask when more secure file access permission checks are implemented.

It has been suggested that trusted systems developers might appreciate ameliorating the requirement that the mode mask ‘‘affects’’ the file access permissions, since it seems access control lists might replace the mode mask to some degree. The wording has been changed to say that it affects the file permission bits, and it leaves the details of the behavior of how they affect the file access permissions to the description in the System Interfaces volume of IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, the System Interfaces volume of IEEE Std 1003.1-200x, *umask*( )

CHANGE HISTORY

First released in Issue 2.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The octal mode is supported.

NAME

unalias — remove alias definitions

SYNOPSIS

UP
```
unalias alias-name...

unalias -a
```

DESCRIPTION

The *unalias* utility shall remove the definition for each alias name specified. See Section 2.3.1 (on page 2234). The aliases shall be removed from the current shell execution environment; see Section 2.12 (on page 2263).

OPTIONS

The *unalias* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−a	Remove all alias definitions from the current shell execution environment.

OPERANDS

The following operand shall be supported:

alias-name	The name of an alias to be removed.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *unalias*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	XSI Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	One of the *alias-name* operands specified did not represent a valid alias definition, or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *unalias* affects the current shell execution environment, it is generally provided as a shell regular built-in.

EXAMPLES

None.

RATIONALE

The *unalias* description is based on that from historical KornShell implementations. Known differences exist between that and the C shell. The KornShell version was adopted to be consistent with all the other KornShell features in this volume of IEEE Std 1003.1-200x, such as command line editing.

The −**a** option is the equivalent of the *unalias* * form of the C shell and is provided to address security concerns about unknown aliases entering the environment of a user (or application) through the allowable implementation-defined predefined alias route or as a result of an *ENV* file. (Although *unalias* could be used to simplify the ''secure'' shell script shown in the *command* rationale, it does not obviate the need to quote all command names. An initial call to *unalias* −**a** would have to be quoted in case there was an alias for *unalias*.)

FUTURE DIRECTIONS

None.

SEE ALSO

alias

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

NAME

uname — return system name

SYNOPSIS

```
uname [−snrvma]
```

DESCRIPTION

By default, the *uname* utility shall write the operating system name to standard output. When options are specified, symbols representing one or more system characteristics shall be written to the standard output. The format and contents of the symbols are implementation-defined. On systems conforming to the System Interfaces volume of IEEE Std 1003.1-200x, the symbols written shall be those supported by the *uname*( ) function as defined in the System Interfaces volume of IEEE Std 1003.1-200x.

OPTIONS

The *uname* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a	Behave as though all of the options **−mnrsv** were specified.
−m	Write the name of the hardware type on which the system is running to standard output.
−n	Write the name of this node within an implementation-defined communications network.
−r	Write the current release level of the operating system implementation.
−s	Write the name of the implementation of the operating system.
−v	Write the current version level of this release of the operating system implementation.

If no options are specified, the *uname* utility shall write the operating system name, as if the **−s** option had been specified.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uname*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

By default, the output shall be a single line of the following form:

```
"%s\n", <sysname>
```

If the −**a** option is specified, the output shall be a single line of the following form:

```
"%s %s %s %s %s\n", <sysname>, <nodename>, <release>,
    <version>, <machine>
```

Additional implementation-defined symbols may be written; all such symbols shall be written at the end of the line of output before the <newline>.

If options are specified to select different combinations of the symbols, only those symbols shall be written, in the order shown above for the −**a** option. If a symbol is not selected for writing, its corresponding trailing <blank>s also shall not be written.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The requested information was successfully written.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Note that any of the symbols could include embedded <space>s, which may affect parsing algorithms if multiple options are selected for output.

The node name is typically a name that the system uses to identify itself for intersystem communication addressing.

EXAMPLES

The following command:

```
uname −sr
```

writes the operating system name and release level, separated by one or more <blank>s.

RATIONALE

It was suggested that this utility cannot be used portably since the format of the symbols is implementation-defined. The POSIX.1 working group could not achieve consensus on defining these formats in the underlying *uname*( ) function, and there was no expectation that this volume of IEEE Std 1003.1-200x would be any more successful. Some applications may still find this historical utility of value. For example, the symbols could be used for system log entries or for comparison with operator or user input.

FUTURE DIRECTIONS

None.

SEE ALSO

The System Interfaces volume of IEEE Std 1003.1-200x, *uname*( )

CHANGE HISTORY

First released in Issue 2.

NAME

uncompress — expand compressed data

SYNOPSIS

XSI `uncompress [−cfv][file...]`

DESCRIPTION

The *uncompress* utility shall restore files to their original state after they have been compressed using the *compress* utility. If no files are specified, the standard input shall be uncompressed to the standard output. If the invoking process has appropriate privileges, the ownership, modes, access time, and modification time of the original file shall be preserved.

This utility shall support the uncompressing of any files produced by the *compress* utility on the same implementation. For files produced by *compress* on other systems, *uncompress* supports 9 to 14-bit compression (see *compress* (on page 2465), **−b**); it is implementation-defined whether values of **−b** greater than 14 are supported.

OPTIONS

The *uncompress* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c Write to standard output; no files are changed.

−f Do not prompt for overwriting files. Except when run in the background, if **−f** is not given the user shall be prompted as to whether an existing file should be overwritten. If the standard input is not a terminal and **−f** is not given, *uncompress* shall write a diagnostic message to standard error and exit with a status greater than zero.

−v Write messages to standard error concerning the expansion of each file.

OPERANDS

The following operand shall be supported:

file A pathname of a file. If *file* already has the **.Z** suffix specified, it shall be used as the input file and the output file shall be named **file** with the **.Z** suffix removed. Otherwise, *file* shall be used as the name of the output file and **file** with the **.Z** suffix appended shall be used as the input file.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'−'`.

INPUT FILES

Input files shall be in the format produced by the *compress* utility.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uncompress*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When there are no *file* operands or the **–c** option is specified, the uncompressed output is written to standard output.

STDERR

Prompts shall be written to the standard error output under the conditions specified in the DESCRIPTION and OPTIONS sections. The prompts shall contain the *file* pathname, but their format is otherwise unspecified. Otherwise, the standard error output shall be used only for diagnostic messages.

OUTPUT FILES

Output files are the same as the respective input files to *compress*.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

The input file remains unmodified.

APPLICATION USAGE

The limit of 14 on the *compress* **–b** *bits* argument is to achieve portability to all systems (within the restrictions imposed by the lack of an explicit published file format). Some implementations based on 16-bit architectures cannot support 15 or 16-bit uncompression.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

compress, *zcat*

CHANGE HISTORY

First released in Issue 4.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

unexpand — convert spaces to tabs

SYNOPSIS

UP `unexpand [ −a| −t tablist][file...]`

DESCRIPTION

The *unexpand* utility shall copy files or standard input to standard output, converting <blank>s at the beginning of each line into the maximum number of <tab>s followed by the minimum number of <space>s needed to fill the same column positions originally filled by the translated <blank>s. By default, tabstops shall be set at every eighth column position. Each <backspace> shall be copied to the output, and shall cause the column position count for tab calculations to be decremented; the count shall never be decremented to a value less than one.

OPTIONS

The *unexpand* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−a In addition to translating <blank>s at the beginning of each line, translate all sequences of two or more <blank>s immediately preceding a tab stop to the maximum number of <tab>s followed by the minimum number of <space>s needed to fill the same column positions originally filled by the translated <blank>s.

−t *tablist* Specify the tab stops. The application shall ensure that the *tablist* option-argument is a single argument consisting of a single positive decimal integer or multiple positive decimal integers, separated by <blank>s or commas, in ascending order. If a single number is given, tabs shall be set *tablist* column positions apart instead of the default 8. If multiple numbers are given, the tabs shall be set at those specific column positions.

The application shall ensure that each tab-stop position *N* is an integer value greater than zero, and the list shall be in strictly ascending order. This is taken to mean that, from the start of a line of output, tabbing to position *N* shall cause the next character output to be in the (*N*+1)th column position on that line. When the **−t** option is not specified, the default shall be the equivalent of specifying **−t 8** (except for the interaction with **−a**, described below).

No <space>-to-<tab> conversions shall occur for characters at positions beyond the last of those specified in a multiple tab-stop list.

When **−t** is specified, the presence or absence of the **−a** option shall be ignored; conversion shall not be limited to the processing of leading <blank>s.

OPERANDS

The following operand shall be supported:

file A pathname of a text file to be used as input.

STDIN

See the INPUT FILES section.

INPUT FILES

The input files shall be text files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *unexpand*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files), the processing of <tab>s and <space>s and for the determination of the width in column positions each character would occupy on an output device.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be equivalent to the input files with the specified <space>-to-<tab> conversions. |

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

One non-intuitive aspect of *unexpand* is its restriction to leading spaces when neither **–a** nor **–t** is specified. Users who desire to always convert all spaces in a file can easily alias *unexpand* to use the **–a** or **–t 8** option.

EXAMPLES

None.

RATIONALE

On several occasions, consideration was given to adding a **–t** option to the *unexpand* utility to complement the **–t** in *expand* (see *expand* (on page 2627)). The historical intent of *unexpand* was to translate multiple <blank>s into tab stops, where tab stops were a multiple of eight column positions on most UNIX systems. An early proposal omitted **–t** because it seemed outside the scope of the UPE; it was not described in any of the base documents. However, hard-coding tab stops every eight columns was not suitable for the international community and broke historical precedents for some vendors in the FORTRAN community, so **–t** was restored in conjunction with the list of valid extension categories considered by the standard developers. Thus, *unexpand* is now the logical converse of *expand*.

FUTURE DIRECTIONS

None.

SEE ALSO

expand, *tabs*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The definition of the *LC_CTYPE* environment variable is changed to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

unget — undo a previous get of an SCCS file (**DEVELOPMENT**)

SYNOPSIS

XSI
```
unget [−ns][−r SID] file...
```

DESCRIPTION

The *unget* utility shall reverse the effect of a *get* **−e** done prior to creating the intended new delta.

OPTIONS

The *unget* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−r *SID*	Uniquely identify which delta is no longer intended. (This would have been specified by *get* as the new delta.) The use of this option is necessary only if two or more outstanding *get* commands for editing on the same SCCS file were done by the same person (login name).
−s	Suppress the writing to standard output of the intended delta's SID.
−n	Retain the file that was obtained by *get*, which would normally be removed from the current directory.

OPERANDS

The following operands shall be supported:

file	A pathname of an existing SCCS file or a directory. If *file* is a directory, the *unget* utility shall behave as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files shall be silently ignored. If exactly one *file* operand appears, and it is `'−'`, the standard input shall be read; each line of the standard input shall be taken to be the name of an SCCS file to be processed. Non-SCCS files and unreadable files shall be silently ignored.

STDIN

The standard input shall be a text file used only when the *file* operand is specified as `'−'`. Each line of the text file shall be interpreted as an SCCS pathname.

INPUT FILES

Any SCCS files processed shall be files of an unspecified format. |

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *unget*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall consist of a line for each file, in the following format:

```
"%s\n", <SID removed from file>
```

If there is more than one named file or if a directory or standard input is named, each pathname shall be written before each of the preceding lines:

```
"\n%s:\n", <pathname>
```

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

Any SCCS files updated shall be files of an unspecified format. During processing of a *file*, a locking *z-file*, as described in *get*, and a *q-file* (a working copy of the *p-file*), may be created and deleted. The *p-file* and *g-file*, as described in *get*, shall be deleted. |

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delta, *get*, *sact*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

NAME

uniq — report or filter out repeated lines in a file

SYNOPSIS

```
uniq [−c|−d|−u][−f fields][−s char][input_file [output_file]]
```

DESCRIPTION

The *uniq* utility shall read an input file comparing adjacent lines, and writes one copy of each input line on the output. The second and succeeding copies of repeated adjacent input lines shall not be written.

Repeated lines in the input shall not be detected if they are not adjacent.

OPTIONS

The *uniq* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	Precede each output line with a count of the number of times the line occurred in the input.
−d	Suppress the writing of lines that are not repeated in the input.
−f *fields*	Ignore the first *fields* fields on each input line when doing comparisons, where *fields* is a positive decimal integer. A field is the maximal string matched by the basic regular expression: `[[:blank:]]*[^[:blank:]]*` If the *fields* option-argument specifies more fields than appear on an input line, a null string shall be used for comparison.
−s *chars*	Ignore the first *chars* characters when doing comparisons, where *chars* shall be a positive decimal integer. If specified in conjunction with the **−f** option, the first *chars* characters after the first *fields* fields shall be ignored. If the *chars* option-argument specifies more characters than remain on an input line, a null string shall be used for comparison.
−u	Suppress the writing of lines that are repeated in the input.

OPERANDS

The following operands shall be supported:

input_file	A pathname of the input file. If the *input_file* operand is not specified, or if the *input_file* is `'−'`, the standard input shall be used.
output_file	A pathname of the output file. If the *output_file* operand is not specified, the standard output shall be used. The results are unspecified if the file named by *output_file* is the file named by *input_file*.

STDIN

The standard input shall be used only if no *input_file* operand is specified or if *input_file* is `'−'`. See the INPUT FILES section.

INPUT FILES

The input file shall be a text file.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uniq*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) which characters constitute a <blank> in the current locale.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be used only if no *output_file* operand is specified. See the OUTPUT FILES section.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

If the **−c** option is specified, the application shall ensure that the output file is empty or each line shall be of the form:

`"%d %s", <number of duplicates>, <line>`

otherwise, the application shall ensure that the output file is empty or each line shall be of the form:

`"%s", <line>`

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	The utility executed successfully.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *sort* utility can be used to cause repeated lines to be adjacent in the input file.

EXAMPLES

The following input file data (but flushed left) was used for a test series on *uniq*:

```
#01 foo0 bar0 foo1 bar1
#02 bar0 foo1 bar1 foo1
#03 foo0 bar0 foo1 bar1
#04
#05 foo0 bar0 foo1 bar1
#06 foo0 bar0 foo1 bar1
#07 bar0 foo1 bar1 foo0
```

What follows is a series of test invocations of the *uniq* utility that use a mixture of *uniq* options against the input file data. These tests verify the meaning of *adjacent*. The *uniq* utility views the input data as a sequence of strings delimited by `'\n'`. Accordingly, for the *fields*th member of the sequence, *uniq* interprets unique or repreated adjacent lines strictly relative to the *fields*+1th member.

1. This first example tests the line counting option, comparing each line of the input file data starting from the second field:

   ```
   uniq −c −f 1 uniq_0I.t
       1 #01 foo0 bar0 foo1 bar1
       1 #02 bar0 foo1 bar1 foo0
       1 #03 foo0 bar0 foo1 bar1
       1 #04
       2 #05 foo0 bar0 foo1 bar1
       1 #07 bar0 foo1 bar1 foo0
   ```

   The number '2', prefixing the fifth line of output, signifies that the *uniq* utility detected a pair of repeated lines. Given the input data, this can only be true when *uniq* is run using the **−f 1** option (which shall cause *uniq* to ignore the first field on each input line).

2. The second example tests the option to suppress unique lines, comparing each line of the input file data starting from the second field:

   ```
   uniq −d −f 1 uniq_0I.t
   #05 foo0 bar0 foo1 bar1
   ```

3. This test suppresses repeated lines, comparing each line of the input file data starting from the second field:

   ```
   uniq −u −f 1 uniq_0I.t
   #01 foo0 bar0 foo1 bar1
   #02 bar0 foo1 bar1 foo1
   #03 foo0 bar0 foo1 bar1
   #04
   #07 bar0 foo1 bar1 foo0
   ```

4. This suppresses unique lines, comparing each line of the input file data starting from the third character:

   ```
   uniq −d −s 2 uniq_0I.t
   ```

   In the last example, the *uniq* utility found no input matching the above criteria.

RATIONALE

Some historical implementations have limited lines to be 1 080 bytes in length, which does not meet the implied {LINE_MAX} limit.

FUTURE DIRECTIONS

None.

SEE ALSO

comm, *sort*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The obsolescent SYNOPSIS and associated text are removed.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

unlink — call the *unlink*( ) function

SYNOPSIS

XSI

```
unlink file
```

DESCRIPTION

The *unlink* utility shall perform the function call:

```
unlink(file);
```

A user may need appropriate privilege to invoke the *unlink* utility.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

file	The pathname of an existing file.

STDIN

Not used.

INPUT FILES

Not used.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *unlink*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

link, *rm*, the System Interfaces volume of IEEE Std 1003.1-200x, *unlink*( )

CHANGE HISTORY

First released in Issue 5.

NAME

uucp — system-to-system copy

SYNOPSIS

XSI
```
uucp [−cCdfjmr][−n user] source-file... destination-file
```

DESCRIPTION

The *uucp* utility shall copy files named by the *source-file* arguments to the *destination-file* argument. The files named can be on local or remote systems.

The *uucp* utility cannot guarantee support for all character encodings in all circumstances. For example, transmission data may be restricted to 7 bits by the underlying network, 8-bit data and filenames need not be portable to non-internationalized systems, and so on. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646: 1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used, and that only characters defined in the portable filename character set be used for naming files. The protocol for transfer of files is unspecified by IEEE Std 1003.1-200x.

Typical implementations of this utility require a communications line configured to use the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility shall write an error message describing the problem and exit with a non-zero exit status.

OPTIONS

The *uucp* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	Do not copy local file to the spool directory for transfer to the remote machine (default).
−C	Force the copy of local files to the spool directory for transfer.
−d	Make all necessary directories for the file copy (default).
−f	Do not make intermediate directories for the file copy.
−j	Write the job identification string to standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.
−m	Send mail to the requester when the copy is completed.
−n *user*	Notify *user* on the remote system that a file was sent.
−r	Do not start the file transfer; just queue the job.

OPERANDS

The following operands shall be supported:

destination-file, *source-file*

A pathname of a file to be copied to, or from, respectively. Either name can be a pathname on the local machine, or can have the form:

```
system-name!pathname
```

where *system-name* is taken from a list of system names that *uucp* knows about. The destination *system-name* can also be a list of names such as:

`system-name!system-name!...!system-name!pathname`

in which case, an attempt is made to send the file via the specified route to the destination. Care should be taken to ensure that intermediate nodes in the route are willing to forward information.

The shell pattern matching notation characters `'?'`, `'*'`, and `"[...]"` appearing in *pathname* shall be expanded on the appropriate system. |

Pathnames can be one of:

1. An absolute pathname.
2. A pathname preceded by ˜*user* where *user* is a login name on the specified system and is replaced by that user's login directory. Note that if an invalid login is specified, the default is to the public directory (called *PUBDIR*; the actual location of *PUBDIR* is implementation-defined).
3. A pathname preceded by ˜/*destination* where *destination* is appended to *PUBDIR*.

   Note: This destination is treated as a filename unless more than one file is being transferred by this request or the destination is already a directory. To ensure that it is a directory, follow the destination with a `'/'`. For example, **˜/dan/** as the destination makes the directory **PUBDIR/dan** if it does not exist and put the requested files in that directory.
4. Anything else shall be prefixed by the current directory. |

If the result is an erroneous pathname for the remote system, the copy shall fail. If the *destination-file* is a directory, the last part of the *source-file* name shall be used. |

The read, write, and execute permissions given by *uucp* are implementation-defined.

STDIN

Not used.

INPUT FILES

The files to be copied are regular files.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uucp*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE
Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements within bracketed filename patterns.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes within bracketed filename patterns (for example, `"'[[:lower:]]*'"`).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output files (which may be on other systems) are copies of the input files.

If the **–m** is used, mail files are modified.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The domain of remotely accessible files can (and for obvious security reasons usually should) be severely restricted.

Note that the '!' character in addresses has to be escaped when using *csh* as a command interpreter because of its history substitution syntax. For *ksh* and *sh* the escape is not necessary, but may be used.

As noted above, shell metacharacters appearing in pathnames are expanded on the appropriate system. On an internationalized system, this is done under the control of local settings of *LC_COLLATE* and *LC_CTYPE*. Thus, care should be taken when using bracketed filename patterns, as collation and typing rules may vary from one system to another. Also be aware that certain types of expression (that is, equivalence classes, character classes, and collating symbols) need not be supported on non-internationalized systems.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *uuencode*, *uustat*, *uux*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section.

The UN margin codes and associated shading are removed from the **−C**, **−f**, **−j**, **−n**, and **−r** options in response to The Open Group Base Resolution bwg2001-003.

NAME

uudecode — decode a binary file

SYNOPSIS

UP
```
uudecode [-o outfile][file]
```

DESCRIPTION

The *uudecode* utility shall read a file, or standard input if no file is specified, that includes data created by the *uuencode* utility. The *uudecode* utility shall scan the input file, searching for data compatible with one of the formats specified in *uuencode* and attempt to create or overwrite the file described by the data (or overridden by the **−o** option). The pathname shall be contained in the data or specified by the **−o** option. The file access permission bits and contents for the file to be produced shall be contained in that data. The mode bits of the created file (other than standard output) shall be set from the file access permission bits contained in the data; that is, other attributes of the mode, including the file mode creation mask (see *umask*), shall not affect the file being produced.

If the pathname of the file to be produced exists, and the user does not have write permission on that file, *uudecode* shall terminate with an error. If the pathname of the file to be produced exists, and the user has write permission on that file, the existing file shall be overwritten.

If the input data was produced by *uuencode* on a system with a different number of bits per byte than on the target system, the results of *uudecode* are unspecified.

OPTIONS

The *uudecode* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported by the implementation:

−o *outfile* A pathname of a file that shall be used instead of any pathname contained in the input data. Specifying an *outfile* option-argument of **/dev/stdout** shall indicate standard output.

OPERANDS

The following operand shall be supported:

file The pathname of a file containing the output of *uuencode*.

STDIN

See the INPUT FILES section.

INPUT FILES

The input files shall be files containing the output of *uuencode*.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uudecode*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the file data header encoded by *uuencode* is – or **/dev/stdout**, or the –**o** **/dev/stdout** option overrides the file data, the standard output shall be in the same format as the file originally encoded by *uuencode*. Otherwise, the standard output shall not be used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The output file shall be in the same format as the file originally encoded by *uuencode*.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The user who is invoking *uudecode* must have write permission on any file being created.

The output of *uuencode* is essentially an encoded bit stream that is not cognizant of byte boundaries. It is possible that a 9-bit byte target machine can process input from an 8-bit source, if it is aware of the requirement, but the reverse is unlikely to be satisfying. Of course, the only data that is meaningful for such a transfer between architectures is generally character data.

EXAMPLES

None.

RATIONALE

Input files are not necessarily text files, as stated by an early proposal. Although the *uuencode* output is a text file, that output could have been wrapped within another file or mail message that is not a text file.

The –**o** option is not historical practice, but was added at the request of WG15 so that the user could override the target pathname without having to edit the input data itself.

In early drafts, the [–**o** *outfile*] option-argument allowed the use of – to mean standard output. The symbol – has only been used previously in IEEE Std 1003.1-200x as a standard input indicator. The developers of the standard did not wish to overload the meaning of – in this manner. The **/dev/stdout** concept exists on most modern systems. The **/dev/stdout** syntax does not refer to a new special file. It is just a magic cookie to specify standard output.

FUTURE DIRECTIONS

None.

SEE ALSO

uuencode

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The −**o** *outfile* option is added, as specified in the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

uuencode — encode a binary file

SYNOPSIS

UP

```
uuencode [−m][file] decode_pathname
```

DESCRIPTION

The *uuencode* utility shall write an encoded version of the named input file, or standard input if no *file* is specified, to standard output. The output shall be encoded using one of the algorithms described in the STDOUT section and shall include the file access permission bits (in *chmod* octal or symbolic notation) of the input file and the *decode_pathname*, for re-creation of the file on another system that conforms to this volume of IEEE Std 1003.1-200x.

OPTIONS

The *uuencode* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported by the implementation:

−m Encode the output using the MIME Base64 algorithm described in STDOUT. If **−m** is not specified, the historical algorithm described in STDOUT shall be used.

OPERANDS

The following operands shall be supported:

decode_pathname
The pathname of the file into which the *uudecode* utility shall place the decoded file. Specifying a *decode_pathname* operand of **/dev/stdout** shall indicate that *uudecode* is to use standard output. If there are characters in *decode_pathname* that are not in the portable filename character set the results are unspecified.

file A pathname of the file to be encoded.

STDIN

See the INPUT FILES section.

INPUT FILES

Input files can be files of any type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uuencode*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

uuencode Base64 Algorithm

The standard output shall be a text file (encoded in the character set of the current locale) that begins with the line:

```
"begin-base64Δ%sΔ%s\n", <mode>, <decode_pathname>
```

and ends with the line:

```
"====\n"
```

In both cases, the lines shall have no preceding or trailing <blank>s.

The encoding process represents 24-bit groups of input bits as output strings of four encoded characters. Proceeding from left to right, a 24-bit input group shall be formed by concatenating three 8-bit input groups. Each 24-bit input group then shall be treated as four concatenated 6-bit groups, each of which shall be translated into a single digit in the base64 alphabet. When encoding a bit stream via the base64 encoding, the bit stream shall be presumed to be ordered with the most-significant bit first. That is, the first bit in the stream shall be the high-order bit in the first byte, and the eighth bit shall be the low-order bit in the first byte, and so on. Each 6-bit group is used as an index into an array of 64 printable characters, as shown in Table 4-21.

Table 4-21 uuencode Base64 Values

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

The character referenced by the index shall be placed in the output string.

The output stream (encoded bytes) shall be represented in lines of no more than 76 characters each. All line breaks or other characters not found in the table shall be ignored by decoding software (see *uudecode*).

Special processing shall be performed if fewer than 24 bits are available at the end of a message or encapsulated part of a message. A full encoding quantum shall always be completed at the end of a message. When fewer than 24 input bits are available in an input group, zero bits shall be added (on the right) to form an integral number of 6-bit groups. Output character positions that are not required to represent actual input data shall be set to the character `'='`. Since all base64 input is an integral number of octets, only the following cases can arise:

1. The final quantum of encoding input is an integral multiple of 24 bits; here, the final unit of encoded output shall be an integral multiple of 4 characters with no `'='` padding.
2. The final quantum of encoding input is exactly 16 bits; here, the final unit of encoded output shall be three characters followed by one `'='` padding character.
3. The final quantum of encoding input is exactly 8 bits; here, the final unit of encoded output shall be two characters followed by two `'='` padding characters.

A terminating `"===="` evaluates to nothing and denotes the end of the encoded data.

uuencode Historical Algorithm

The standard output shall be a text file (encoded in the character set of the current locale) that begins with the line:

`"beginΔ%sΔ%s\n"` *<mode>*, *<decode_pathname>*

and ends with the line:

`"end\n"`

In both cases, the lines shall have no preceding or trailing <blank>s.

The algorithm that shall be used for lines in between **begin** and **end** takes three octets as input and writes four characters of output by splitting the input at six-bit intervals into four octets, containing data in the lower six bits only. These octets shall be converted to characters by adding a value of 0x20 to each octet, so that each octet is in the range [0x20,0x5f], and then it shall be assumed to represent a printable character in the ISO/IEC 646:1991 standard encoded character set. It then shall be translated into the corresponding character codes for the codeset in use in the current locale. (For example, the octet 0x41, representing `'A'`, would be translated to `'A'` in the current codeset, such as 0xc1 if it were EBCDIC.)

Where the bits of two octets are combined, the least significant bits of the first octet shall be shifted left and combined with the most significant bits of the second octet shifted right. Thus the three octets *A*, *B*, *C* shall be converted into the four octets:

```
0x20 + (( A >> 2                    ) & 0x3F)
0x20 + (((A << 4) | ((B >> 4) & 0xF)) & 0x3F)
0x20 + (((B << 2) | ((C >> 6) & 0x3)) & 0x3F)
0x20 + (( C                         ) & 0x3F)
```

These octets then shall be translated into the local character set.

Each encoded line contains a length character, equal to the number of characters to be decoded plus 0x20 translated to the local character set as described above, followed by the encoded characters. The maximum number of octets to be encoded on each line shall be 45.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The file is expanded by 35 percent (each three octets become four, plus control information) causing it to take longer to transmit.

Since this utility is intended to create files to be used for data interchange between systems with possibly different codesets, and to represent binary data as a text file, the ISO/IEC 646:1991 standard was chosen for a midpoint in the algorithm as a known reference point. The output from *uuencode* is a text file on the local system. If the output were in the ISO/IEC 646:1991 standard codeset, it might not be a text file (at least because the <newline>s might not match), and the goal of creating a text file would be defeated. If this text file was then carried to another machine with the same codeset, it would be perfectly compatible with that system's *uudecode*. If it was transmitted over a mail system or sent to a machine with a different codeset, it is assumed that, as for every other text file, some translation mechanism would convert it (by the time it reached a user on the other system) into an appropriate codeset. This translation only makes sense from the local codeset, not if the file has been put into a ISO/IEC 646:1991 standard representation first. Similarly, files processed by *uuencode* can be placed in *pax* archives, intermixed with other text files in the same codeset.

EXAMPLES

None.

RATIONALE

A new algorithm was added at the request of the international community to parallel work in RFC 2045 (MIME). As with the historical *uuencode* format, the Base64 Content-Transfer-Encoding is designed to represent arbitrary sequences of octets in a form that is not humanly readable. A 65-character subset of the ISO/IEC 646:1991 standard is used, enabling 6 bits to be represented per printable character. (The extra 65th character, `'='`, is used to signify a special processing function.)

This subset has the important property that it is represented identically in all versions of the ISO/IEC 646:1991 standard, including US ASCII, and all characters in the subset are also represented identically in all versions of EBCDIC. The historical *uuencode* algorithm does not share this property, which is the reason that a second algorithm was added to the ISO POSIX-2 standard.

The string `"===="` was used for the termination instead of the end used in the original format because the latter is a string that could be valid encoded input.

In an early draft, the −**m** option was named −**b** (for Base64), but it was renamed to reflect its relationship to the RFC 2045. A −**u** was also present to invoke the default algorithm, but since this was not historical practice, it was omitted as being unnecessary.

See the RATIONALE section in *uudecode* for the derivation of the **/dev/stdout** symbol.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *uudecode*

CHANGE HISTORY

First released in Issue 4.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The Base64 algorithm and the ability to output to **/dev/stdout** are added as specified in the IEEE P1003.2b draft standard.

NAME

uustat — uucp status inquiry and job control

SYNOPSIS

XSI
```
uustat [ −q| −k jobid| −r jobid]

uustat [−s system][−u user]
```

DESCRIPTION

The *uustat* utility shall display the status of, or cancel, previously specified *uucp* requests, or provide general status on *uucp* connections to other systems.

When no options are given, *uustat* shall write to standard output the status of all *uucp* requests issued by the current user.

Typical implementations of this utility require a communications line configured to use the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility shall write an error message describing the problem and exit with a non-zero exit status.

OPTIONS

The *uustat* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−q	Write the jobs queued for each machine.
−k *jobid*	Kill the *uucp* request whose job identification is *jobid*. The application shall ensure that the killed *uucp* request belongs to the person invoking *uustat* unless that user has appropriate privileges.
−r *jobid*	Rejuvenate *jobid*. The files associated with *jobid* are touched so that their modification time is set to the current time. This prevents the cleanup program from deleting the job until the jobs modification time reaches the limit imposed by the program.
−s *system*	Write the status of all *uucp* requests for remote system *system*.
−u *user*	Write the status of all *uucp* requests issued by *user*.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uustat*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall consist of information about each job selected, in an unspecified format. The information shall include at least the job ID, the user ID or name, and the remote system name.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

uucp

CHANGE HISTORY

First released in Issue 2.

Issue 6

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The *LC_TIME* and *TZ* entries are removed from the ENVIRONMENT VARIABLES section. |

The UN margin code and associated shading are removed from the −**q** option in reaponse to The Open Group Base Resolution bwg2001-003. |

NAME

uux — remote command execution

SYNOPSIS

XSI
```
uux [−np] command−string

uux [−jnp] command−string
```

DESCRIPTION

The *uux* utility shall gather zero or more files from various systems, execute a shell pipeline (see Section 2.9 (on page 2248)) on a specified system, and then send the standard output of the command to a file on a specified system. Only the first command of a pipeline can have a *system-name*! prefix. All other commands in the pipeline shall be executed on the system of the first command.

The following restrictions are applicable to the shell pipeline processed by *uux*:

- In gathering files from different systems, pathname expansion shall not be performed by *uux*. Thus, a request such as:

  ```
  uux "c99 remsys!˜/*.c"
  ```

  would attempt to copy the file named literally ***.c** to the local system.

- The redirection operators `">>"`, `"<<"`, `">|"`, and `">&"` shall not be accepted. Any use of these redirection operators shall cause this utility to write an error message describing the problem and exit with a non-zero exit status.

- The reserved word **!** cannot be used at the head of the pipeline to modify the exit status. (See the *command-string* operand description below.)

- Alias substitution shall not be performed.

A filename can be specified as for *uucp*; it can be an absolute pathname, a pathname preceded by ~*name* (which is replaced by the corresponding login directory), a pathname specified as ~/*dest*(*dest* is prefixed by the public directory called *PUBDIR*; the actual location of *PUBDIR* is implementation-defined), or a simple filename (which is prefixed by *uux* with the current directory). See *uucp* (on page 3163) for the details.

The execution of commands on remote systems shall take place in an execution directory known to the *uucp* system. All files required for the execution shall be put into this directory unless they already reside on that machine. Therefore, the application shall ensure that non-local filenames (without path or machine reference) are unique within the *uux* request.

The *uux* utility shall attempt to get all files to the execution system. For files that are output files, the application shall ensure that the filename is escaped using parentheses.

The remote system shall notify the user by mail if the requested command on the remote system was disallowed or the files were not accessible. This notification can be turned off by the **−n** option.

Typical implementations of this utility require a communications line configured to use the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility shall write an error message describing the problem and exits with a non-zero exit status.

The *uux* utility cannot guarantee support for all character encodings in all circumstances. For example, transmission data may be restricted to 7 bits by the underlying network, 8-bit data and filenames need not be portable to non-internationalized systems, and so on. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used and that only characters defined in the portable filename character set be used for naming files.

OPTIONS

The *uux* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−p	Make the standard input to *uux* the standard input to the *command-string*.
−j	Write the job identification string to standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.
−n	Do not notify the user if the command fails.

OPERANDS

The following operand shall be supported:

command-string
A string made up of one or more arguments that are similar to normal command arguments, except that the command and any filenames can be prefixed by *system-name*!. A null *system-name* shall be interpreted as the local system.

STDIN

The standard input shall not be used unless the '−' or **−p** option is specified; in those cases, the standard input shall be made the standard input of the *command-string*.

INPUT FILES

Input files shall be selected according to the contents of *command-string*.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *uux*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall be not used unless the **−j** option is specified; in that case, the job identification string shall be written to standard output in the following format:

```
"%s\n", <jobid>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

Output files shall be created or written, or both, according to the contents of *command-string*.

If the **−n** is not used, mail files shall be modified following any command or file-access failures on the remote system.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Note that, for security reasons, many installations limit the list of commands executable on behalf of an incoming request from *uux*. Many sites permit little more than the receipt of mail via *uux*.

Any characters special to the command interpreter should be quoted either by quoting the entire *command-string* or quoting the special characters as individual arguments.

As noted in *uucp*, shell pattern matching notation characters appearing in pathnames are expanded on the appropriate local system. This is done under the control of local settings of *LC_COLLATE* and *LC_CTYPE*. Thus, care should be taken when using bracketed filename patterns, as collation and typing rules may vary from one system to another. Also be aware that certain types of expression (that is, equivalence classes, character classes, and collating symbols) need not be supported on non-internationalized systems.

EXAMPLES

1. The following command gets **file1** from system **a** and **file2** file from system **b**, executes *diff* on the local system, and puts the results in **file.diff** in the local *PUBDIR* directory. (*PUBDIR* is the *uucp* public directory on the local system.)

   ```
   uux "!diff a!/usr/file1 b!/a4/file2 >!˜/file.diff"
   ```

2. The following command fails because *uux* places all files copied to a system in the same working directory. Although the files **xyz** are from two different systems, their filenames are the same and conflict.

   ```
   uux "!diff a!/usr1/xyz b!/usr2/xyz >!˜/xyz.diff"
   ```

3. The following command succeeds (assuming *diff* is permitted on system **a**) because the file local to system **a** is not copied to the working directory, and hence does not conflict the file from system **c**.

```
uux "a!diff a!/usr/xyz c!/usr/xyz >!~/xyz.diff"
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

uucp, *uuencode*, *uustat*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The obsolescent SYNOPSIS is removed.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

The UN margin code and associated shading are removed from the **–j** option in response to The Open Group Base Resolution bwg2001-003.

NAME

val — validate SCCS files (**DEVELOPMENT**)

SYNOPSIS

XSI
```
val -

val [-s][-m name][-r SID][-y type] file...
```

DESCRIPTION

The *val* utility shall determine whether the specified *file* is an SCCS file meeting the characteristics specified by the options.

OPTIONS

The *val* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines, except that the usage of the `'−'` operand is not strictly as intended by the guidelines (that is, reading options and operands from standard input).

The following options shall be supported:

−**m** *name*	Specify a *name*, which is compared with the SCCS %**M**% keyword in *file*; see *get* (on page 2675).
−**r** *SID*	Specify a *SID* (SCCS Identification String), an SCCS delta number. A check shall be made to determine whether the *SID* is ambiguous (for example, −**r 1** is ambiguous because it physically does not exist but implies 1.1, 1.2, and so on, which may exist) or invalid (for example, −**r 1.0** or −**r 1.1.0** are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check shall be made to determine whether it actually exists.
−**s**	Silence the diagnostic message normally written to standard output for any error that is detected while processing each named file on a given command line.
−**y** *type*	Specify a *type*, which shall be compared with the SCCS %**Y**% keyword in *file*; see *get* (on page 2675).

OPERANDS

The following operands shall be supported:

file	A pathname of an existing SCCS file. If exactly one *file* operand appears, and it is `'−'`, the standard input shall be read: each line shall be independently processed as if it were a command line argument list. (However, the line is not subjected to any of the shell word expansions, such as parameter expansion or quote removal.) |

STDIN

The standard input shall be a text file used only when the *file* operand is specified as `'−'`.

INPUT FILES

Any SCCS files processed shall be files of an unspecified format. |

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *val*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall consist of informative messages about either:

1. Each file processed
2. Each command line read from standard input

If the standard input is not used, for each *file* operand yielding a discrepancy, the output line shall have the following format:

`"%s: %s\n",` *`<pathname>, <unspecified string>`*

If standard input is used, a line of input shall be written before each of the preceding lines for files containing discrepancies:

`"%s:\n",` *`<input line>`*

STDERR

Not used.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The 8-bit code returned by *val* shall be a disjunction of the possible errors, that is, it can be interpreted as a bit string where set bits are interpreted as follows:

0x80 = Missing file argument.
0x40 = Unknown or duplicate option.
0x20 = Corrupted SCCS file.
0x10 = Cannot open file or file not SCCS.
0x08 = *SID* is invalid or ambiguous.
0x04 = *SID* does not exist.
0x02 = %**Y**%, −**y** mismatch.
0x01 = %**M**%, −**m** mismatch.

Note that *val* can process two or more files on a given command line and can process multiple command lines (when reading the standard input). In these cases an aggregate code shall be returned: a logical OR of the codes generated for each command line and file processed.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since the *val* exit status sets the 0x80 bit, shell applications checking `"$?"` cannot tell if it terminated due to a missing file argument or receipt of a signal.

EXAMPLES

In a directory with three SCCS files, **s.x** (of **t** type ‘‘text’’), **s.y**, and **s.z** (a corrupted file), the following command could produce the output shown:

```
val − <<EOF
−y source s.x
−m y s.y
s.z
EOF
```

```
−y source s.x

    s.x: %Y%, −y mismatch
s.z

    s.z: corrupted SCCS file
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

admin, *delta*, *get*, *prs*

CHANGE HISTORY

First released in Issue 2.

Issue 6

The Open Group Corrigendum U025/4 is applied, correcting a typographical error in the EXIT STATUS.

The normative text is reworded to emphasize the term ‘‘shall’’ for implementation requirements.

NAME

vi — screen-oriented (visual) display editor

SYNOPSIS

UP
```
vi [−rR][−l][−c command][−t tagstring][−w size][file ...]
```

DESCRIPTION

This utility shall be provided on systems that both support the User Portability Utilities option and define the POSIX2_CHAR_TERM symbol. On other systems it is optional.

The *vi* (visual) utility is a screen-oriented text editor. Only the open and visual modes of the editor are described in IEEE Std 1003.1-200x; see the line editor *ex* for additional editing capabilities used in *vi*. The user can switch back and forth between *vi* and *ex* and execute *ex* commands from within *vi*.

This reference page uses the term *edit buffer* to describe the current working text. No specific implementation is implied by this term. All editing changes are performed on the edit buffer, and no changes to it shall affect any file until an editor command writes the file.

When using *vi*, the terminal screen acts as a window into the editing buffer. Changes made to the editing buffer shall be reflected in the screen display; the position of the cursor on the screen shall indicate the position within the editing buffer.

Certain terminals do not have all the capabilities necessary to support the complete *vi* definition. When these commands cannot be supported on such terminals, this condition shall not produce an error message such as ''not an editor command'' or report a syntax error. The implementation may either accept the commands and produce results on the screen that are the result of an unsuccessful attempt to meet the requirements of this volume of IEEE Std 1003.1-200x or report an error describing the terminal-related deficiency.

OPTIONS

The *vi* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−**c** *command*	See the *ex* command description of the −**c** option.
−**r**	See the *ex* command description of the −**r** option.
−**R**	See the *ex* command description of the −**R** option.
−**t** *tagstring*	See the *ex* command description of the −**t** option.
−**w** *size*	See the *ex* command description of the −**w** option.

OPERANDS

See the OPERANDS section of the *ex* command for a description of the operands supported by the *vi* command.

STDIN

If standard input is not a terminal device, the results are undefined. The standard input consists of a series of commands and input text, as described in the EXTENDED DESCRIPTION section.

If a read from the standard input returns an error, or if the editor detects an end-of-file condition from the standard input, it shall be equivalent to a SIGHUP asynchronous event.

INPUT FILES

See the INPUT FILES section of the *ex* command for a description of the input files supported by the *vi* command.

ENVIRONMENT VARIABLES

See the ENVIRONMENT VARIABLES section of the *ex* command for the environment variables that affect the execution of the *vi* command.

ASYNCHRONOUS EVENTS

See the ASYNCHRONOUS EVENTS section of the *ex* for the asynchronous events that affect the execution of the *vi* command.

STDOUT

If standard output is not a terminal device, undefined results occur.

Standard output may be used for writing prompts to the user, for informational messages, and for writing lines from the file.

STDERR

If standard output is not a terminal device, undefined results occur.

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

See the OUTPUT FILES section of the *ex* command for a description of the output files supported by the *vi* command.

EXTENDED DESCRIPTION

If the terminal does not have the capabilities necessary to support an unspecified portion of the *vi* definition, implementations shall start initially in *ex* mode or open mode. Otherwise, after initialization, *vi* shall be in command mode; text input mode can be entered by one of several commands used to insert or change text. In text input mode, <ESC> can be used to return to command mode; other uses of <ESC> are described later in this section; see **Terminate Command or Input Mode** (on page 3194).

Initialization in ex and vi

See **Initialization in ex and vi** (on page 2559) for a description of *ex* and *vi* initialization for the *vi* utility.

Command Descriptions in vi

The following symbols are used in this reference page to represent arguments to commands.

buffer See the description of *buffer* in the EXTENDED DESCRIPTION section of the *ex* utility; see **Command Descriptions in ex** (on page 2569).

In open and visual mode, when a command synopsis shows both [*buffer*] and [*count*] preceding the command name, they can be specified in either order.

count A positive integer used as an optional argument to most commands, either to give a repeat count or as a size. This argument is optional and shall default to 1 unless otherwise specified.

The Synopsis lines for the *vi* commands <control>-G, <control>-L, <control>-R, <control>-], **%**, **&**, **^**, **D**, **m**, **M**, **Q**, **u**, **U**, and **ZZ** do not have *count* as an optional argument. Regardless, it shall not be an error to specify a *count* to these commands, and any specified *count* shall be ignored.

motion An optional trailing argument used by the **!**, **<**, **>**, **c**, **d**, and **y** commands, which is used to indicate the region of text that shall be affected by the command. The motion can be either one of the command characters repeated or one of several other *vi* commands (listed in the following table). Each of the applicable commands specifies the region of text matched by repeating the command; each command that can be used as a motion command specifies the region of text it affects.

Commands that take *motion* arguments operate on either lines or characters, depending on the circumstances. When operating on lines, all lines that fall partially or wholly within the text region specified for the command shall be affected. When operating on characters, only the exact characters in the specified text region shall be affected. Each motion command specifies this individually.

When commands that may be motion commands are not used as motion commands, they shall set the current position to the current line and column as specified.

The following commands shall be valid cursor motion commands:

`<control>-H`
`<newline>`
`<carriage-return>`
`<control>-N`
`<control>-P`
`<space>`
`$`
`%`
`'`*`character`*
`` ` ``*`character`*
`(`
`)`
`+`
`,`
`-`
`/`
`_`
`;`
`?`
`B`
`E`
`F`
`G`
`H`
`L`
`M`
`N`
`T`
`W`
`[[`
`]]`
`^`
`b`
`e`
`f`
`h`
`j`
`k`
`l`
`n`
`t`
`w`
`{`
`|`
`}`
`0`

Any *count* that is specified to a command that has an associated motion command shall be applied to the motion command. If a *count* is applied to both the command and its associated motion command, the effect shall be multiplicative.

The following symbols are used in this section to specify locations in the edit buffer:

current character
The character that is currently indicated by the cursor.

end of a line
The point located between the last non-<newline> (if any) and the terminating <newline> of a line. For an empty line, this location coincides with the beginning of the line.

end of the edit buffer
The location corresponding to the end of the last line in the edit buffer.

The following symbols are used in this section to specify command actions:

bigword In the POSIX locale, *vi* shall recognize four kinds of *bigwords*:

1. A maximal sequence of non-<blank>s preceded and followed by <blank>s or the beginning or end of a line or the edit buffer
2. One or more sequential blank lines
3. The first character in the edit buffer
4. The last non-<newline> in the edit buffer

word In the POSIX locale, *vi* shall recognize five kinds of words:

1. A maximal sequence of letters, digits, and underscores, delimited at both ends by:
   - Characters other than letters, digits, or underscores
   - The beginning or end of a line
   - The beginning or end of the edit buffer
2. A maximal sequence of characters other than letters, digits, underscores, or <blank>s, delimited at both ends by:
   - A letter, digit, underscore
   - <blank>s
   - The beginning or end of a line
   - The beginning or end of the edit buffer
3. One or more sequential blank lines
4. The first character in the edit buffer
5. The last non-<newline> in the edit buffer

section boundary
A *section boundary* is one of the following:

1. A line whose first character is a <form-feed>
2. A line whose first character is an open curly brace (`'{'`)
3. A line whose first character is a period and whose second and third characters match a two-character pair in the **sections** edit option (see *ed*)
4. A line whose first character is a period and whose only other character matches the first character of a two-character pair in the **sections** edit option, where the second character of the two-character pair is a <space>
5. The first line of the edit buffer
6. The last line of the edit buffer if the last line of the edit buffer is empty or if it is a **]]** or **}** command; otherwise, the last non-<newline> of the last line of the edit buffer

paragraph boundary
A *paragraph boundary* is one of the following:

1. A section boundary
2. A line whose first character is a period and whose second and third characters match a two-character pair in the **paragraphs** edit option (see *ed*)
3. A line whose first character is a period and whose only other character matches the first character of a two-character pair in the *paragraphs* edit option, where the second character of the two-character pair is a <space>
4. One or more sequential blank lines

remembered search direction
See the description of remembered search direction in *ed*.

sentence boundary

A *sentence boundary* is one of the following:

1. A paragraph boundary
2. The first non-<blank> that occurs after a paragraph boundary
3. The first non-<blank> that occurs after a period ('.'), exclamation mark ('!'), or question mark ('?'), followed by two <space>s or the end of a line; any number of closing parenthesis (')'), closing brackets (']'), double quote ('"'), or single quote ('\'') characters can appear between the punctuation mark and the two <space>s or end-of-line

In the remainder of the description of the *vi* utility, the term ‘‘buffer line’’ refers to a line in the edit buffer and the term ‘‘display line’’ refers to the line or lines on the display screen used to display one buffer line. The term ‘‘current line’’ refers to a specific ‘‘buffer line’’.

If there are display lines on the screen for which there are no corresponding buffer lines because they correspond to lines that would be after the end of the file, they shall be displayed as a single tilde ('˜') character, plus the terminating <newline>.

The last line of the screen shall be used to report errors or display informational messages. It shall also be used to display the input for ‘‘line-oriented commands’’ (**/**, **?**, **:**, and **!**). When a line-oriented command is executed, the editor shall enter text input mode on the last line on the screen, using the respective command characters as prompt characters. (In the case of the **!** command, the associated motion shall be entered by the user before the editor enters text input mode.) The line entered by the user shall be terminated by a <newline>, a non-<control>-V-escaped <carriage-return>, or unescaped <ESC>. It is unspecified if more characters than require a display width minus one column number of screen columns can be entered.

If any command is executed that overwrites a portion of the screen other than the last line of the screen (for example, the *ex* **suspend**, or **!** commands), other than the *ex* **shell** command, the user shall be prompted for a character before the screen is refreshed and the edit session continued.

<tab>s shall take up the number of columns on the screen set by the **tabstop** edit option (see *ed*), unless there are less than that number of columns before the display margin that will cause the displayed line to be folded; in this case, they shall only take up the number of columns up to that boundary.

The cursor shall be placed on the current line and relative to the current column as specified by each command described in the following sections.

In open mode, if the current line is not already displayed, then it shall be displayed.

In visual mode, if the current line is not displayed, then the lines that are displayed shall be expanded, scrolled, or redrawn to cause an unspecified portion of the current line to be displayed. If the screen is redrawn, no more than the number of display lines specified by the value of the **window** edit option shall be displayed (unless the current line cannot be completely displayed in the number of display lines specified by the **window** edit option) and the current line shall be positioned as close to the center of the displayed lines as possible (within the constraints imposed by the distance of the line from the beginning or end of the edit buffer). If the current line is before the first line in the display and the screen is scrolled, an unspecified portion of the current line shall be placed on the first line of the display. If the current line is after the last line in the display and the screen is scrolled, an unspecified portion of the current line shall be placed on the last line of the display.

In visual mode, if a line from the edit buffer (other than the current line) does not entirely fit into the lines at the bottom of the display that are available for its presentation, the editor may choose not to display any portion of the line. The lines of the display that do not contain text from the edit buffer for this reason shall each consist of a single `'@'` character.

In visual mode, the editor may choose for unspecified reasons to not update lines in the display to correspond to the underlying edit buffer text. The lines of the display that do not correctly correspond to text from the edit buffer for this reason shall consist of a single `'@'` character (plus the terminating <newline>), and the <control>-R command shall cause the editor to update the screen to correctly represent the edit buffer.

Open and visual mode commands that set the current column set it to a column position in the display, and not a character position in the line. In this case, however, the column position in the display shall be calculated for a infinite width display; for example, the column related to a character that is part of a line that has been folded onto additional screen lines will be offset from the display line column where the buffer line begins, not from the beginning of a particular display line.

The display cursor column in the display is based on the value of the current column, as follows, with each rule applied in turn:

1. If the current column is after the last display line column used by the displayed line, the display cursor column shall be set to the last display line column occupied by the last non-<newline> in the current line; otherwise, the display cursor column shall be set to the current column.
2. If the character of which some portion is displayed in the display line column specified by the display cursor column requires more than a single display line column:
    a. If in text input mode, the display cursor column shall be adjusted to the first display line column in which any portion of that character is displayed.
    b. Otherwise, the display cursor column shall be adjusted to the last display line column in which any portion of that character is displayed.

The current column shall not be changed by these adjustments to the display cursor column.

If an error occurs during the parsing or execution of a *vi* command:

- The terminal shall be alerted. Execution of the *vi* command shall stop, and the cursor (for example, the current line and column) shall not be further modified.
- Unless otherwise specified by the following command sections, it is unspecified whether an informational message shall be displayed.
- Any partially entered *vi* command shall be discarded.
- If the *vi* command resulted from a **map** expansion, all characters from that **map** expansion shall be discarded, except as otherwise specified by the **map** command (see *ed*).
- If the *vi* command resulted from the execution of a buffer, no further commands caused by the execution of the buffer shall be executed.

Page Backwards

Synopsis: `[count] <control>-B`

If in open mode, the <control>-B command shall behave identically to the **z** command. Otherwise, if the current line is the first line of the edit buffer, it shall be an error.

If the **window** edit option is less than 3, display a screen where the last line of the display shall be some portion of:

`(current first line) −1`

otherwise, display a screen where the first line of the display shall be some portion of:

`(current first line) − count x ((window edit option) −2)`

If this calculation would result in a line that is before the first line of the edit buffer, the first line of the display shall display some portion of the first line of the edit buffer.

Current line: If no lines from the previous display remain on the screen, set to the last line of the display; otherwise, set to (*line* – the number of new lines displayed on this screen).

Current column: Set to non-<blank>.

Scroll Forward

Synopsis: `[count] <control>-D`

If the current line is the last line of the edit buffer, it shall be an error.

If no *count* is specified, *count* shall default to the *count* associated with the previous <control>-D or <control>-U command. If there was no previous <control>-D or <control>-U command, *count* shall default to the value of the **scroll** edit option.

If in open mode, write lines starting with the line after the current line, until *count* lines or the last line of the file have been written.

Current line: If the current line + *count* is past the last line of the edit buffer, set to the last line of the edit buffer; otherwise, set to the current line + *count*.

Current column: Set to non-<blank>.

Scroll Forward by Line

Synopsis: `[count] <control>-E`

Display the line count lines after the last line currently displayed.

If the last line of the edit buffer is displayed, it shall be an error. If there is no line *count* lines after the last line currently displayed, the last line of the display shall display some portion of the last line of the edit buffer.

Current line: Unchanged if the previous current character is displayed; otherwise, set to the first line displayed.

Current column: Unchanged.

Page Forward

Synopsis: `[count] <control>-F`

If in open mode, the <control>-F command shall behave identically to the **z** command. Otherwise, if the current line is the last line of the edit buffer, it shall be an error.

If the **window** edit option is less than 3, display a screen where the first line of the display shall be some portion of:

`(current last line) +1`

otherwise, display a screen where the first line of the display shall be some portion of:

`(current first line) + count x ((window edit option) −2)`

If this calculation would result in a line that is after the last line of the edit buffer, the last line of the display shall display some portion of the last line of the edit buffer.

Current line: If no lines from the previous display remain on the screen, set to the first line of the display; otherwise, set to (*line* + the number of new lines displayed on this screen).

Current column: Set to non-<blank>.

Display Information

Synopsis: `<control>-G`

This command shall be equivalent to the *ex* **file** command .

Move Cursor Backwards

Synopsis:

```
[count] <control>-H
[count] h
the current erase character (see stty)
```

If there are no characters before the current character on the current line, it shall be an error. If there are less than *count* previous characters on the current line, *count* shall be adjusted to the number of previous characters on the line.

If used as a motion command:

1. The text region shall be from the character before the starting cursor up to and including the *count*th character before the starting cursor.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to (*column* – the number of columns occupied by *count* characters ending with the previous current column).

Move Down

Synopsis:

```
[count] <newline>
[count] <control>-J
[count] <control>-M
[count] <control>-N
[count] j
[count] <carriage-return>
[count] +
```

If there are less than *count* lines after the current line in the edit buffer, it shall be an error.

If used as a motion command:

1. The text region shall include the starting line and the next *count* – 1 lines.
2. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

Current line: Set to *current line* + *count*.

Current column: Set to non-<blank> for the <carriage-return>, <control>-M, and **+** commands; otherwise, unchanged.

Clear and Redisplay

Synopsis: `<control>-L`

If in open mode, clear the screen and redisplay the current line. Otherwise, clear and redisplay the screen.

Current line: Unchanged.

Current column: Unchanged.

Move Up

Synopsis:

```
[count] <control>-P
[count] k
[count] –
```

If there are less than *count* lines before the current line in the edit buffer, it shall be an error.

If used as a motion command:

1. The text region shall include the starting line and the previous *count* lines.
2. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

Current line: Set to *current line* – *count*.

Current column: Set to non-<blank> for the – command; otherwise, unchanged.

Redraw Screen

Synopsis: `<control>-R`

If any lines have been deleted from the display screen and flagged as deleted on the terminal using the **@** convention (see the beginning of the EXTENDED DESCRIPTION section), they shall be redisplayed to match the contents of the edit buffer.

It is unspecified whether lines flagged with **@** because they do not fit on the terminal display shall be affected.

Current line: Unchanged.

Current column: Unchanged.

Scroll Backward

Synopsis: `[`*count*`] <control>-U`

If the current line is the first line of the edit buffer, it shall be an error.

If no *count* is specified, *count* shall default to the *count* associated with the previous <control>-D or <control>-U command. If there was no previous <control>-D or <control>-U command, *count* shall default to the value of the **scroll** edit option.

Current line: If *count* is greater than the current line, set to 1; otherwise, set to the current line – *count*.

Current column: Set to non-<blank>.

Scroll Backward by Line

Synopsis: `[`*count*`] <control>-Y`

Display the line *count* lines before the first line currently displayed.

If the current line is the first line of the edit buffer, it shall be an error. If this calculation would result in a line that is before the first line of the edit buffer, the first line of the display shall display some portion of the first line of the edit buffer.

Current line: Unchanged if the previous current character is displayed; otherwise, set to the first line displayed.

Current column: Unchanged.

Edit the Alternate File

Synopsis: `<control>-ˆ`

This command shall be equivalent to the *ex* **edit** command, with the alternate pathname as its argument.

Terminate Command or Input Mode

Synopsis: `<ESC>`

If a partial *vi* command (as defined by at least one, non-*count* character) has been entered, discard the *count* and the command character(s).

Otherwise, if no command characters have been entered, and the <ESC> was the result of a map expansion, the terminal shall be alerted and the <ESC> character shall be discarded, but it shall not be an error.

Otherwise, it shall be an error.

Current line: Unchanged.

Current column: Unchanged.

Search for tagstring

Synopsis: `<control>-]`

If the current character is not a word or <blank>, it shall be an error.

This command shall be equivalent to the *ex* **tag** command, with the argument to that command defined as follows.

If the current character is a <blank>:

1. Skip all <blank>s after the cursor up to the end of the line.
2. If the end of the line is reached, it shall be an error.

Then, the argument to the *ex* **tag** command shall be the current character and all subsequent characters, up to the first non-word character or the end of the line.

Move Cursor Forward

Synopsis: **[***count***]** `<space>`
[*count***]** `l` (ell)

If there are less than *count* non-<newline>s after the cursor on the current line, *count* shall be adjusted to the number of non-<newline>s after the cursor on the line.

If used as a motion command:

1. If the current or *count*th character after the cursor is the last non-<newline> in the line, the text region shall be comprised of the current character up to and including the last non-<newline> in the line. Otherwise, the text region shall be from the current character up to, but not including, the *count*th character after the cursor.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

If there are no non-<newline>s after the current character on the current line, it shall be an error.

Current line: Unchanged.

Current column: Set to the last column that displays any portion of the *count*th character after the current character.

Replace Text with Results from Shell Command

Synopsis: **[***count***]** `!` *motion shell-commands* `<newline>`

If the motion command is the **!** command repeated:

1. If the edit buffer is empty and no *count* was supplied, the command shall be the equivalent of the *ex* **:read !** command, with the text input, and no text shall be copied to any buffer.
2. Otherwise:
   a. If there are less than *count* −1 lines after the current line in the edit buffer, it shall be an error.

   b. The text region shall be from the current line up to and including the next *count* −1 lines.

Otherwise, the text region shall be the lines in which any character of the text region specified by the motion command appear.

Any text copied to a buffer shall be in line mode.

This command shall be equivalent to the *ex* **!** command for the specified lines.

Move Cursor to End-of-line

Synopsis: `[count] $`

It shall be an error if there are less than (*count* −1) lines after the current line in the edit buffer.

If used as a motion command:

1. If *count* is 1:
   a. It shall be an error if the line is empty.
   b. Otherwise, the text region shall consist of all characters from the starting cursor to the last non-<newline> in the line, inclusive, and any text copied to a buffer shall be in character mode.
2. Otherwise, if the starting cursor position is at or before the first non-<blank> in the line, the text region shall consist of the current and the next *count* −1 lines, and any text saved to a buffer shall be in line mode.
3. Otherwise, the text region shall consist of all characters from the starting cursor to the last non-<newline> in the line that is *count* −1 lines forward from the current line, and any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the *current line* + *count*−1.

Current column: The current column is set to the last display line column of the last non-<newline> in the line, or column position 1 if the line is empty.

The current column shall be adjusted to be on the last display line column of the last non-<newline> of the current line as subsequent commands change the current line, until a command changes the current column.

Move to Matching Character

Synopsis: `%`

If the character at the current position is not a parenthesis, bracket, or curly brace, search forward in the line to the first one of those characters. If no such character is found, it shall be an error.

The matching character shall be the parenthesis, bracket, or curly brace matching the parenthesis, bracket, or curly brace, respectively, that was at the current position or that was found on the current line.

Matching shall be determined as follows, for an open parenthesis:

1. Set a counter to 1.
2. Search forwards until a parenthesis is found or the end of the edit buffer is reached.

3. If the end of the edit buffer is reached, it shall be an error.
4. If an open parenthesis is found, increment the counter by 1.
5. If a close parenthesis is found, decrement the counter by 1.
6. If the counter is zero, the current character is the matching character.

Matching for a close parenthesis shall be equivalent, except that the search shall be backwards, from the starting character to the beginning of the buffer, a close parenthesis shall increment the counter by 1, and an open parenthesis shall decrement the counter by 1.

Matching for brackets and curly braces shall be equivalent, except that searching shall be done for open and close brackets or open and close curly braces. It is implementation-defined whether other characters are searched for and matched as well.

If used as a motion command:

1. If the matching cursor was after the starting cursor in the edit buffer, and the starting cursor position was at or before the first non-<blank> non-<newline> in the starting line, and the matching cursor position was at or after the last non-<blank> non-<newline> in the matching line, the text region shall consist of the current line to the matching line, inclusive, and any text copied to a buffer shall be in line mode.
2. If the matching cursor was before the starting cursor in the edit buffer, and the starting cursor position was at or after the last non-<blank> non-<newline> in the starting line, and the matching cursor position was at or before the first non-<blank> non-<newline> in the matching line, the text region shall consist of the current line to the matching line, inclusive, and any text copied to a buffer shall be in line mode.
3. Otherwise, the text region shall consist of the starting character to the matching character, inclusive, and any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the line where the matching character is located.

Current column: Set to the last column where any portion of the matching character is displayed.

Repeat Substitution

Synopsis: `&`

Repeat the previous substitution command. This command shall be equivalent to the *ex* **&** command with the current line as its addresses, and without *options*, *count*, or *flags*.

Return to Previous Context at Beginning of Line

Synopsis: `' character`

It shall be an error if there is no line in the edit buffer marked by *character*.

If used as a motion command:

1. If the starting cursor is after the marked cursor, then the locations of the starting cursor and the marked cursor in the edit buffer shall be logically swapped.
2. The text region shall consist of the starting line up to and including the marked line, and any text copied to a buffer shall be in line mode.

If not used as a motion command:

Current line: Set to the line referenced by the mark.

Current column: Set to non-<blank>.

Return to Previous Context

Synopsis: `‘ character`

It shall be an error if the marked line is no longer in the edit buffer. If the marked line no longer contains a character in the saved numbered character position, it shall be as if the marked position is the first non-<blank>.

If used as a motion command:

1. It shall be an error if the marked cursor references the same character in the edit buffer as the starting cursor.
2. If the starting cursor is after the marked cursor, then the locations of the starting cursor and the marked cursor in the edit buffer shall be logically swapped.
3. If the starting line is empty or the starting cursor is at or before the first non-<blank> non-<newline> of the starting line, and the marked cursor line is empty or the marked cursor references the first character of the marked cursor line, the text region shall consist of all lines containing characters from the starting cursor to the line before the marked cursor line, inclusive, and any text copied to a buffer shall be in line mode.
4. Otherwise, if the marked cursor line is empty or the marked cursor references a character at or before the first non-<blank> non-<newline> of the marked cursor line, the region of text shall be from the starting cursor to the last non-<newline> of the line before the marked cursor line, inclusive, and any text copied to a buffer shall be in character mode.
5. Otherwise, the region of text shall be from the starting cursor (inclusive), to the marked cursor (exclusive), and any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the line referenced by the mark.

Current column: Set to the last column in which any portion of the character referenced by the mark is displayed.

Return to Previous Section

Synopsis: `[[`

Move the cursor backward through the edit buffer to the first character of the previous section boundary, *count* times.

If used as a motion command:

1. If the starting cursor was at the first character of the starting line or the starting line was empty, and the first character of the boundary was the first character of the boundary line, the text region shall consist of the current line up to and including the line where the *count*th next boundary starts, and any text copied to a buffer shall be in line mode.
2. If the boundary was the last line of the edit buffer or the last non-<newline> of the last line of the edit buffer, the text region shall consist of the last character in the edit buffer up to and including the starting character, and any text saved to a buffer shall be in character mode.

3. Otherwise, the text region shall consist of the starting character up to but not including the first character in the *count*th next boundary, and any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the line where the *count*th next boundary in the edit buffer starts.

Current column: Set to the last column in which any portion of the first character of the *count*th next boundary is displayed, or column position 1 if the line is empty.

Move to Next Section

Synopsis:]]

Move the cursor forward through the edit buffer to the first character of the next section boundary, *count* times.

If used as a motion command:

1. If the starting cursor was at the first character of the starting line or the starting line was empty, and the first character of the boundary was the first character of the boundary line, the text region shall consist of the current line up to and including the line where the *count*th previous boundary starts, and any text copied to a buffer shall be in line mode.
2. If the boundary was the first line of the edit buffer, the text region shall consist of the first character in the edit buffer up to but not including the starting character, and any text copied to a buffer shall be in character mode.
3. Otherwise, the text region shall consist of the first character in the *count*th previous section boundary up to but not including the starting character, and any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the line where the *count*th previous boundary in the edit buffer starts.

Current column: Set to the last column in which any portion of the first character of the *count*th previous boundary is displayed, or column position 1 if the line is empty.

Move to First Non-<blank> Position on Current Line

Synopsis: ^

If used as a motion command:

1. If the line has no non-<blank> non-<newline>s, or if the cursor is at the first non-<blank> non-<newline> of the line, it shall be an error.
2. If the cursor is before the first non-<blank> non-<newline> of the line, the text region shall be comprised of the current character, up to, but not including, the first non-<blank> non-<newline> of the line.
3. If the cursor is after the first non-<blank> non-<newline> of the line, the text region shall be from the character before the starting cursor up to and including the first non-<blank> non-<newline> of the line.
4. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to non-<blank>.

Current and line above

Synopsis: `[count] _`

If there are less than *count* –1 lines after the current line in the edit buffer, it shall be an error.

If used as a motion command:

1. If *count* is less than 2, the text region shall be the current line.
2. Otherwise, the text region shall include the starting line and the next *count* –1 lines.
3. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

Current line: Set to current line + *count* –1.

Current column: Set to non-<blank>.

Move Back to Beginning of Sentence

Synopsis: `[count] (`

Move backward to the beginning of a sentence. This command shall be equivalent to the **[[** command, with the exception that sentence boundaries shall be used instead of section boundaries.

Move Forward to Beginning of Sentence

Synopsis: `[count] )`

Move forward to the beginning of a sentence. This command shall be equivalent to the **]]** command, with the exception that sentence boundaries shall be used instead of section boundaries.

Move Back to Preceding Paragraph

Synopsis: `[count] {`

Move back to the beginning of the preceding paragraph. This command shall be equivalent to the **[[** command, with the exception that paragraph boundaries shall be used instead of section boundaries.

Move Forward to Next Paragraph

Synopsis: `[count] }`

Move forward to the beginning of the next paragraph. This command shall be equivalent to the **]]** command, with the exception that paragraph boundaries shall be used instead of section boundaries.

Move to Specific Column Position

Synopsis: `[count] |`

For the purposes of this command, lines that are too long for the current display and that have been folded shall be treated as having a single, 1–based, number of columns.

If there are less than *count* columns in which characters from the current line are displayed on the screen, *count* shall be adjusted to be the last column in which any portion of the line is displayed on the screen.

If used as a motion command:

1. If the line is empty, or the cursor character is the same as the character on the *count*th column of the line, it shall be an error.
2. If the cursor is before the *count*th column of the line, the text region shall be comprised of the current character, up to but not including the character on the *count*th column of the line.
3. If the cursor is after the *count*th column of the line, the text region shall be from the character before the starting cursor up to and including the character on the *count*th column of the line.
4. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to the last column in which any portion of the character that is displayed in the *count* column of the line is displayed.

Reverse Find Character

Synopsis: `[count] ,`

If the last **F**, **f**, **T**, or **t** command was **F**, **f**, **T**, or **t**, this command shall be equivalent to an **f**, **F**, **t**, or **T** command, respectively, with the specified *count* and the same search character.

If there was no previous **F**, **f**, **T**, or **t** command, it shall be an error.

Repeat

Synopsis: `[count] .`

Repeat the last **!**, **<**, **>**, **A**, **C**, **D**, **I**, **J**, **O**, **P**, **R**, **S**, **X**, **Y**, **a**, **c**, **d**, **i**, **o**, **p**, **r**, **s**, **x**, **y**, or **˜** command. It shall be an error if none of these commands have been executed. Commands (other than commands that enter text input mode) executed as a result of map expansions, shall not change the value of the last repeatable command.

Repeated commands with associated motion commands shall repeat the motion command as well; however, any specified *count* shall replace the *count*(s) that were originally specified to the repeated command or its associated motion command.

If the motion component of the repeated command is **f**, **F**, **t**, or **T**, the repeated command shall not set the remembered search character for the **;** and **,** commands.

If the repeated command is **p** or **P**, and the buffer associated with that command was a numeric buffer named with a number less than 9, the buffer associated with the repeated command shall be set to be the buffer named by the name of the previous buffer logically incremented by 1.

If the repeated character is a text input command, the input text associated with that command is repeated literally:

- Input characters are neither macro or abbreviation-expanded.
- Input characters are not interpreted in any special way with the exception that <newline>, <carriage-return>, and <control>-T behave as described in **Input Mode Commands in vi** (on page 3220).

Current line: Set as described for the repeated command.

Current column: Set as described for the repeated command.

Find Regular Expression

Synopsis: /

If the input line contains no non-<newline>s, it shall be equivalent to a line containing only the last regular expression encountered. The enhanced regular expressions supported by *vi* are described in **Regular Expressions in ex** (on page 2592).

Otherwise, the line shall be interpreted as one or more regular expressions, optionally followed by an address offset or a *vi* **z** command.

If the regular expression is not the last regular expression on the line, or if a line offset or **z** command is specified, the regular expression shall be terminated by an unescaped '/' character, which shall not be used as part of the regular expression. If the regular expression is not the first regular expression on the line, it shall be preceded by zero or more <blank>s, a semicolon, zero or more <blank>s, a leading '/' character, which shall not be interpreted as part of the regular expression. It shall be an error to precede any regular expression with any characters other than these.

Each search shall begin from the character after the first character of the last match (or, if it is the first search, after the cursor). If the **wrapscan** edit option is set, the search shall continue to the character before the starting cursor character; otherwise, to the end of the edit buffer. It shall be an error if any search fails to find a match, and an informational message to this effect shall be displayed.

An optional address offset (see **Addressing in ex** (on page 2562)) can be specified after the last regular expression by including a trailing '/' character after the regular expression and specifying the address offset. This offset will be from the line containing the match for the last regular expression specified. It shall be an error if the line offset would indicate a line address less than 1 or greater than the last line in the edit buffer. An address offset of zero shall be supported. It shall be an error to follow the address offset with any other characters than <blank>s.

If not used as a motion command, an optional **z** command (see **Redraw Window** (on page 3219)) can be specified after the last regular expression by including a trailing '/' character after the regular expression, zero or more <blank>s, a 'z', zero or more <blank>s, an optional new **window** edit option value, zero or more <blank>s, and a location character. The effect shall be as if the **z** command was executed after the / command. It shall be an error to follow the **z** command with any other characters than <blank>s.

The remembered search direction shall be set to forward.

If used as a motion command:

1. It shall be an error if the last match references the same character in the edit buffer as the starting cursor.

2. If any address offset is specified, the last match shall be adjusted by the specified offset as described previously.
3. If the starting cursor is after the last match, then the locations of the starting cursor and the last match in the edit buffer shall be logically swapped.
4. If any address offset is specified, the text region shall consist of all lines containing characters from the starting cursor to the last match line, inclusive, and any text copied to a buffer shall be in line mode.
5. Otherwise, if the starting line is empty or the starting cursor is at or before the first non-<blank> non-<newline> of the starting line, and the last match line is empty or the last match starts at the first character of the last match line, the text region shall consist of all lines containing characters from the starting cursor to the line before the last match line, inclusive, and any text copied to a buffer shall be in line mode.
6. Otherwise, if the last match line is empty or the last match begins at a character at or before the first non-<blank> non-<newline> of the last match line, the region of text shall be from the current cursor to the last non-<newline> of the line before the last match line, inclusive, and any text copied to a buffer shall be in character mode.
7. Otherwise, the region of text shall be from the current cursor (inclusive), to the first character of the last match (exclusive), and any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: If a match is found, set to the last matched line plus the address offset, if any; otherwise, unchanged.

Current column: Set to the last column on which any portion of the first character in the last matched string is displayed, if a match is found; otherwise, unchanged.

Move to First Character in Line

Synopsis: `0` (zero)

Move to the first character on the current line. The character `'0'` shall not be interpreted as a command if it is immediately preceded by a digit.

If used as a motion command:

1. If the cursor character is the first character in the line, it shall be an error.
2. The text region shall be from the character before the cursor character up to and including the first character in the line.
3. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: The last column in which any portion of the first character in the line is displayed, or if the line is empty, unchanged.

Execute an ex Command

Synopsis: `:`

Execute one or more *ex* commands.

If any portion of the screen other than the last line of the screen was overwritten by any *ex* command (except **shell**), *vi* shall display a message indicating that it is waiting for an input from the user, and shall then read a character. This action may also be taken for other, unspecified reasons.

If the next character entered is a `':'`, another *ex* command shall be accepted and executed. Any other character shall cause the screen to be refreshed and *vi* shall return to command mode.

Current line: As specified for the *ex* command.

Current column: As specified for the *ex* command.

Repeat Find

Synopsis: `[count] ;`

This command shall be equivalent to the last **F**, **f**, **T**, or **t** command, with the specified *count*, and with the same search character used for the last **F**, **f**, **T**, or **t** command. If there was no previous **F**, **f**, **T**, or **t** command, it shall be an error.

Shift Left

Synopsis: `[count] < motion`

If the motion command is the **<** command repeated:

1. If there are less than *count* −1 lines after the current line in the edit buffer, it shall be an error.
2. The text region shall be from the current line, up to and including the next *count* −1 lines.

Shift any line in the text region specified by the *count* and motion command one shiftwidth (see the *ex* **shiftwidth** option) toward the start of the line, as described by the *ex* **<** command. The unshifted lines shall be copied to the unnamed buffer in line mode.

Current line: If the motion was from the current cursor position toward the end of the edit buffer, unchanged. Otherwise, set to the first line in the edit buffer that is part of the text region specified by the motion command.

Current column: Set to non-<blank>.

Shift Right

Synopsis: `[count] > motion`

If the motion command is the **>** command repeated:

1. If there are less than *count* −1 lines after the current line in the edit buffer, it shall be an error.
2. The text region shall be from the current line, up to and including the next *count* −1 lines.

Shift any line with characters in the text region specified by the *count* and motion command one shiftwidth (see the *ex* **shiftwidth** option) away from the start of the line, as described by the *ex* **>** command. The unshifted lines shall be copied into the unnamed buffer in line mode.

Current line: If the motion was from the current cursor position toward the end of the edit buffer, unchanged. Otherwise, set to the first line in the edit buffer that is part of the text region specified by the motion command.

Current column: Set to non-<blank>.

Scan Backwards for Regular Expression

Synopsis: `?`

Scan backwards; The **?** command shall be equivalent to the **/** command (see **Find Regular Expression** (on page 3202)) with the following exceptions:

1. The input prompt shall be a `'?'`.
2. Each search shall begin from the character before the first character of the last match (or, if it is the first search, the character before the cursor character).
3. The search direction shall be from the cursor toward the beginning of the edit buffer, and the **wrapscan** edit option shall affect whether the search wraps to the end of the edit buffer and continues.
4. The remembered search direction shall be set to backward.

Execute

Synopsis: `@buffer`

If the *buffer* is specified as **@**, the last buffer executed shall be used. If no previous buffer has been executed, it shall be an error.

Behave as if the contents of the named buffer were entered as standard input. After each line of a line-mode buffer, and all but the last line of a character mode buffer, behave as if a <newline> were entered as standard input.

If an error occurs during this process, an error message shall be written, and no more characters resulting from the execution of this command shall be processed.

If a *count* is specified, behave as if that count were entered as user input before the characters from the **@** buffer were entered.

Current line: As specified for the individual commands.

Current column: As specified for the individual commands.

Reverse Case

Synopsis: `[count] ~`

Reverse the case of the current character and the next *count* −1 characters, such that lowercase characters that have uppercase counterparts shall be changed to uppercase characters, and uppercase characters that have lowercase counterparts shall be changed to lowercase characters, as prescribed by the current locale. No other characters shall be affected by this command.

If there are less than *count* −1 characters after the cursor in the edit buffer, *count* shall be adjusted to the number of characters after the cursor in the edit buffer minus 1.

For the purposes of this command, the next character after the last non-<newline> on the line shall be the next character in the edit buffer.

Current line: Set to the line including the (*count*−1)th character after the cursor.

Current column: Set to the last column in which any portion of the (*count*−1)th character after the cursor is displayed.

Append

Synopsis: `[count] a`

Enter text input mode after the current cursor position. No characters already in the edit buffer shall be affected by this command. A *count* shall cause the input text to be appended *count* −1 more times to the end of the input.

Current line/column: As specified for the text input commands (see **Input Mode Commands in vi** (on page 3220)).

Append at End-of-Line

Synopsis: `[count] A`

This command shall be equivalent to the *vi* command:

```
$ [ count ] a
```

(see **Append**).

Move Backward to Preceding Word

Synopsis: `[count] b`

With the exception that words are used as the delimiter instead of bigwords, this command shall be equivalent to the **B** command.

Move Backward to Preceding Bigword

Synopsis: `[count] B`

If the edit buffer is empty or the cursor is on the first character of the edit buffer, it shall be an error. If less than *count* bigwords begin between the cursor and the start of the edit buffer, *count* shall be adjusted to the number of bigword beginnings between the cursor and the start of the edit buffer.

If used as a motion command:

1. The text region shall be from the first character of the *count*th previous bigword beginning up to but not including the cursor character.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the line containing the *current column*.

Current column: Set to the last column upon which any part of the first character of the *count*th previous bigword is displayed.

Change

Synopsis: `[buffer][count] c motion`

If the motion command is the **c** command repeated:

1. The buffer text shall be in line mode.
2. If there are less than *count* −1 lines after the current line in the edit buffer, it shall be an error.
3. The text region shall be from the current line up to and including the next *count* −1 lines.

Otherwise, the buffer text mode and text region shall be as specified by the motion command.

The replaced text shall be copied into *buffer*, if specified, and into the unnamed buffer. If the text to be replaced contains characters from more than a single line, or the buffer text is in line mode, the replaced text shall be copied into the numeric buffers as well.

If the buffer text is in line mode:

1. Any lines that contain characters in the region shall be deleted, and the editor shall enter text input mode at the beginning of a new line which shall replace the first line deleted.
2. If the **autoindent** edit option is set, **autoindent** characters equal to the **autoindent** characters on the first line deleted shall be inserted as if entered by the user.

Otherwise, if characters from more than one line are in the region of text:

1. The text shall be deleted.
2. Any text remaining in the last line in the text region shall be appended to the first line in the region, and the last line in the region shall be deleted.
3. The editor shall enter text input mode after the last character not deleted from the first line in the text region, if any; otherwise, on the first column of the first line in the region.

Otherwise:

1. If the glyph for `'$'` is smaller than the region, the end of the region shall be marked with a `'$'`.
2. The editor shall enter text input mode, overwriting the region of text.

Current line/column: As specified for the text input commands (see **Input Mode Commands in vi** (on page 3220)).

Change to End-of-Line

Synopsis: `[buffer][count] C`

This command shall be equivalent to the *vi* command:

`[buffer][count] c$`

See the **c** command.

Delete

Synopsis: `[buffer][count] d motion`

If the motion command is the **d** command repeated:

1. The buffer text shall be in line mode.
2. If there are less than *count* –1 lines after the current line in the edit buffer, it shall be an error.
3. The text region shall be from the current line up to and including the next *count* –1 lines.

Otherwise, the buffer text mode and text region shall be as specified by the motion command.

If in open mode, and the current line is deleted, and the line remains on the display, an `'@'` character shall be displayed as the first glyph of that line.

Delete the region of text into *buffer*, if specified, and into the unnamed buffer. If the text to be deleted contains characters from more than a single line, or the buffer text is in line mode, the deleted text shall be copied into the numeric buffers, as well.

Current line: Set to the first text region line that appears in the edit buffer, unless that line has been deleted, in which case it shall be set to the last line in the edit buffer, or line 1 if the edit buffer is empty.

Current column:

1. If the line is empty, set to column position 1.
2. Otherwise, if the buffer text is in line mode or the motion was from the cursor toward the end of the edit buffer:
   a. If a character from the current line is displayed in the current column, set to the last column that displays any portion of that character.
   b. Otherwise, set to the last column in which any portion of any character in the line is displayed.
3. Otherwise, if a character is displayed in the column that began the text region, set to the last column that displays any portion of that character.
4. Otherwise, set to the last column in which any portion of any character in the line is displayed.

Delete to End-of-Line

Synopsis: `[buffer] D`

Delete the text from the current position to the end of the current line; equivalent to the *vi* command:

```
[buffer] d$
```

Move to End-of-Word

Synopsis: `[count] e`

With the exception that words are used instead of bigwords as the delimiter, this command shall be equivalent to the **E** command.

Move to End-of-Bigword

Synopsis: `[count] E`

If the edit buffer is empty it shall be an error. If less than *count* bigwords end between the cursor and the end of the edit buffer, *count* shall be adjusted to the number of bigword endings between the cursor and the end of the edit buffer.

If used as a motion command:

1. The text region shall be from the last character of the *count*th next bigword up to and including the cursor character.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Set to the line containing the current column.

Current column: Set to the last column upon which any part of the last character of the *count*th next bigword is displayed.

Find Character in Current Line (Forward)

Synopsis: `[count] f character`

It shall be an error if *count* occurrences of the character do not occur after the cursor in the line.

If used as a motion command:

1. The text range shall be from the cursor character up to and including the *count*th occurrence of the specified character after the cursor.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to the last column in which any portion of the *count*th occurrence of the specified character after the cursor appears in the line.

Find Character in Current Line (Reverse)

Synopsis: `[count] F character`

It shall be an error if *count* occurrences of the character do not occur before the cursor in the line.

If used as a motion command:

1. The text region shall be from the *count*th occurrence of the specified character before the cursor, up to, but not including the cursor character.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to the last column in which any portion of the *count*th occurrence of the specified character before the cursor appears in the line.

Move to Line

Synopsis: `[count] G`

If *count* is not specified, it shall default to the last line of the edit buffer. If *count* is greater than the last line of the edit buffer, it shall be an error.

If used as a motion command:

1. The text region shall be from the cursor line up to and including the specified line.
2. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

Current line: Set to *count* if *count* is specified; otherwise, the last line.

Current column: Set to non-<blank>.

Move to Top of Screen

Synopsis: `[count] H`

If the beginning of the line *count* greater than the first line of which any portion appears on the display does not exist, it shall be an error.

If used as a motion command:

1. If in open mode, the text region shall be the current line.
2. Otherwise, the text region shall be from the starting line up to and including (the first line of the display + *count* –1).
3. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

If in open mode, this command shall set the current column to non-<blank> and do nothing else.

Otherwise, it shall set the current line and current column as follows.

Current line: Set to (the first line of the display + *count* –1).

Current column: Set to non-<blank>.

Insert Before Cursor

Synopsis: `[count] i`

Enter text input mode before the current cursor position. No characters already in the edit buffer shall be affected by this command. A *count* shall cause the input text to be appended *count* –1 more times to the end of the input.

Current line/column: As specified for the text input commands (see **Input Mode Commands in vi** (on page 3220)).

Insert at Beginning of Line

Synopsis: `[count] I`

This command shall be equivalent to the *vi* command ^[*count*]**i** command.

Join

Synopsis: `[count] J`

If the current line is the last line in the edit buffer, it shall be an error.

This command shall be equivalent to the *ex* **join** command with no addresses, and an *ex* command *count* value of 1 if *count* was not specified or if a *count* of 1 was specified, and an *ex* command *count* value of *count* −1 for any other value of *count*, except that the current line and column shall be set as follows.

Current line: Unchanged.

Current column: The last column in which any portion of the character following the last character in the initial line is displayed, or the last non-<newline> in the line if no characters were appended.

Move to Bottom of Screen

Synopsis: `[count] L`

If the beginning of the line count less than the last line of which any portion appears on the display does not exist, it shall be an error.

If used as a motion command:

1. If in open mode, the text region shall be the current line.
2. Otherwise, the text region shall include all lines from the starting cursor line to (the last line of the display −(*count* −1)).
3. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

1. If in open mode, this command shall set the current column to non-<blank> and do nothing else.
2. Otherwise, it shall set the current line and current column as follows.

Current line: Set to (the last line of the display −(*count* −1)).

Current column: Set to non-<blank>.

Mark Position

Synopsis: `m letter`

This command shall be equivalent to the *ex* **mark** command with the specified character as an argument.

Move to Middle of Screen

Synopsis: `M`

The middle line of the display shall be calculated as follows:

`(the top line of the display) + (((number of lines displayed) +1) /2) −1`

If used as a motion command:

1. If in open mode, the text region shall be the current line.
2. Otherwise, the text region shall include all lines from the starting cursor line up to and including the middle line of the display.
3. Any text copied to a buffer shall be in line mode.

If not used as a motion command:

If in open mode, this command shall set the current column to non-<blank> and do nothing else.

Otherwise, it shall set the current line and current column as follows.

Current line: Set to the middle line of the display.

Current column: Set to non-<blank>.

Repeat Regular Expression Find (Forward)

Synopsis: `n`

If the remembered search direction was forward, the **n** command shall be equivalent to the *vi* **/** command with no characters entered by the user. Otherwise, it shall be equivalent to the *vi* **?** command with no characters entered by the user.

If the **n** command is used as a motion command for the **!** command, the editor shall not enter text input mode on the last line on the screen, and shall behave as if the user entered a single `'!'` character as the text input.

Repeat Regular Expression Find (Reverse)

Synopsis: `N`

Scan for the next match of the last pattern given to **/** or **?**, but in the reverse direction; this is the reverse of **n**.

If the remembered search direction was forward, the **N** command shall be equivalent to the *vi* **?** command with no characters entered by the user. Otherwise, it shall be equivalent to the *vi* **/** command with no characters entered by the user. If the **N** command is used as a motion command for the **!** command, the editor shall not enter text input mode on the last line on the screen, and shall behave as if the user entered a single **!** character as the text input.

Insert Empty Line Below

Synopsis: `o`

Enter text input mode in a new line appended after the current line. A *count* shall cause the input text to be appended *count* −1 more times to the end of the already added text, each time starting on a new, appended line.

Current line/column: As specified for the text input commands (see **Input Mode Commands in vi** (on page 3220)).

Insert Empty Line Above

Synopsis: `O`

Enter text input mode in a new line inserted before the current line. A *count* shall cause the input text to be appended *count* −1 more times to the end of the already added text, each time starting on a new, appended line.

Current line/column: As specified for the text input commands (see **Input Mode Commands in vi** (on page 3220)).

Put from Buffer Following

Synopsis: `[`*`buffer`*`] p`

If no *buffer* is specified, the unnamed buffer shall be used.

If the buffer text is in line mode, the text shall be appended below the current line, and each line of the buffer shall become a new line in the edit buffer. A *count* shall cause the buffer text to be appended *count* −1 more times to the end of the already added text, each time starting on a new, appended line.

If the buffer text is in character mode, the text shall be appended into the current line after the cursor, and each line of the buffer other than the first and last shall become a new line in the edit buffer. A *count* shall cause the buffer text to be appended *count* −1 more times to the end of the already added text, each time starting after the last added character.

Current line: If the buffer text is in line mode, set the line to line +1; otherwise, unchanged.

Current column: If the buffer text is in line mode:

1. If there is a non-<blank> in the first line of the buffer, set to the last column on which any portion of the first non-<blank> in the line is displayed.
2. If there is no non-<blank> in the first line of the buffer, set to the last column on which any portion of the last non-<newline> in the first line of the buffer is displayed.

If the buffer text is in character mode:

1. If the text in the buffer is from more than a single line, then set to the last column on which any portion of the first character from the buffer is displayed.
2. Otherwise, if the buffer is the unnamed buffer, set to the last column on which any portion of the last character from the buffer is displayed.
3. Otherwise, set to the first column on which any portion of the first character from the buffer is displayed.

Put from Buffer Before

Synopsis: `[`*`buffer`*`] P`

If no *buffer* is specified, the unnamed buffer shall be used.

If the buffer text is in line mode, the text shall be inserted above the current line, and each line of the buffer shall become a new line in the edit buffer. A *count* shall cause the buffer text to be appended *count* −1 more times to the end of the already added text, each time starting on a new, appended line.

If the buffer text is in character mode, the text shall be inserted into the current line before the cursor, and each line of the buffer other than the first and last shall become a new line in the edit buffer. A *count* shall cause the buffer text to be appended *count* −1 more times to the end of the already added text, each time starting after the last added character.

Current line: Unchanged.

Current column: If the buffer text is in line mode:

1. If there is a non-<blank> in the first line of the buffer, set to the last column on which any portion of that character is displayed.
2. If there is no non-<blank> in the first line of the buffer, set to the last column on which any portion of the last non-<newline> in the first line of the buffer is displayed.

If the buffer text is in character mode:

1. If the buffer is the unnamed buffer, set to the last column on which any portion of the last character from the buffer is displayed.
2. Otherwise, set to the first column on which any portion of the first character from the buffer is displayed.

Enter ex Mode

Synopsis: `Q`

Leave visual or open mode and enter *ex* command mode.

Current line: Unchanged.

Current column: Unchanged.

Replace Character

Synopsis: `[count] r character`

Replace the *count* characters at and after the cursor with the specified character. If there are less than *count* non-<newline>s at and after the cursor on the line, it shall be an error.

If character is <control>-V, any next character other than the <newline> shall be stripped of any special meaning and used as a literal character.

If character is <ESC>, no replacement shall be made and the current line and current column shall be unchanged.

If character is <carriage-return> or <newline>, *count* new lines shall be appended to the current line. All but the last of these lines shall be empty. *count* characters at and after the cursor shall be discarded, and any remaining characters after the cursor in the current line shall be moved to the last of the new lines. If the **autoindent** edit option is set, they shall be preceded by the same number of **autoindent** characters found on the line from which the command was executed.

Current line: Unchanged unless the replacement character is a <carriage-return> or <newline>, in which case it shall be set to line + *count*.

Current column: Set to the last column position on which a portion of the last replaced character is displayed, or if the replacement character caused new lines to be created, set to non-<blank>.

Replace Characters

Synopsis: `R`

Enter text input mode at the current cursor position possibly replacing text on the current line. A *count* shall cause the input text to be appended *count* −1 more times to the end of the input. |

Current line/column: As specified for the text input commands (see **Input Mode Commands in vi** (on page 3220)).

Substitute Character

Synopsis: `[buffer][count] s`

This command shall be equivalent to the *vi* command:

`[buffer][count] c<space>`

Substitute Lines

Synopsis: `[buffer][count] S`

This command shall be equivalent to the *vi* command:

`[buffer][count] c_`

Move Cursor to Before Character (Forward)

Synopsis: `[count] t character`

It shall be an error if *count* occurrences of the character do not occur after the cursor in the line.

If used as a motion command:

1. The text region shall be from the cursor up to but not including the *count*th occurrence of the specified character after the cursor.
2. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to the last column in which any portion of the character before the *count*th occurrence of the specified character after the cursor appears in the line.

Move Cursor to After Character (Reverse)

Synopsis: `[count] T character`

It shall be an error if *count* occurrences of the character do not occur before the cursor in the line.

If used as a motion command:

1. If the character before the cursor is the specified character, it shall be an error.
2. The text region shall be from the character before the cursor up to but not including the *count*th occurrence of the specified character before the cursor.
3. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

Current line: Unchanged.

Current column: Set to the last column in which any portion of the character after the *count*th occurrence of the specified character before the cursor appears in the line.

Undo

Synopsis: `u`

This command shall be equivalent to the *ex* **undo** command except that the current line and current column shall be set as follows:

Current line: Set to the first line added or changed if any; otherwise, move to the line preceding any deleted text if one exists; otherwise, move to line 1.

Current column: If undoing an *ex* command, set to the first non-<blank>.

Otherwise, if undoing a text input command:

1. If the command was an **C**, **c**, **O**, **o**, **R**, **S**, or **s** command, the current column shall be set to the value it held when the text input command was entered.
2. Otherwise, set to the last column in which any portion of the first character after the deleted text is displayed, or, if no non-<newline>s follow the text deleted from this line, set to the last column in which any portion of the last non-<newline> in the line is displayed, or 1 if the line is empty.

Otherwise, if a single line was modified (that is, not added or deleted) by the **u** command:

1. If text was added or changed, set to the last column in which any portion of the first character added or changed is displayed.
2. If text was deleted, set to the last column in which any portion of the first character after the deleted text is displayed, or, if no non-<newline>s follow the deleted text, set to the last column in which any portion of the last non-<newline> in the line is displayed, or 1 if the line is empty.

Otherwise, set to non-<blank>.

Undo Current Line

Synopsis: `U`

Restore the current line to its state immediately before the most recent time that it became the current line.

Current line: Unchanged.

Current column: Set to the first column in the line in which any portion of the first character in the line is displayed.

Move to Beginning of Word

Synopsis: `[count] w`

With the exception that words are used as the delimiter instead of bigwords, this command shall be equivalent to the **W** command.

Move to Beginning of Bigword

Synopsis: `[count] W`

If the edit buffer is empty, it shall be an error. If there are less than *count* bigwords between the cursor and the end of the edit buffer, *count* shall be adjusted to move the cursor to the last bigword in the edit buffer.

If used as a motion command:

1. If the associated command is **c**, *count* is 1, and the cursor is on a <blank>, the region of text shall be the current character and no further action shall be taken.
2. If there are less than *count* bigwords between the cursor and the end of the edit buffer, then the command shall succeed, and the region of text shall include the last character of the edit buffer.
3. If there are <blank>s or an end-of-line that precede the *count*th bigword, and the associated command is **c**, the region of text shall be up to and including the last character before the preceding <blank>s or end-of-line.
4. If there are <blank>s or an end-of-line that precede the bigword, and the associated command is **d** or **y**, the region of text shall be up to and including the last <blank> the start of the bigword or end-of-line.
5. Any text copied to a buffer shall be in character mode.

If not used as a motion command:

1. If the cursor is on the last character of the edit buffer, it shall be an error.

Current line: Set to the line containing the current column.

Current column: Set to the last column in which any part of the first character of the *count*th next bigword is displayed.

Delete Character at Cursor

Synopsis: `[buffer][count] x`

Delete the *count* characters at and after the current character into *buffer*, if specified, and into the unnamed buffer.

If the line is empty, it shall be an error. If there are less than *count* non-<newline>s at and after the cursor on the current line, *count* shall be adjusted to the number of non-<newline>s at and after the cursor.

Current line: Unchanged.

Current column: If the line is empty, set to column position 1. Otherwise, if there were *count* or less non-<newline>s at and after the cursor on the current line, set to the last column that displays any part of the last non-<newline> of the line. Otherwise, unchanged.

Delete Character Before Cursor

Synopsis: `[buffer][count] X`

Delete the *count* characters before the current character into *buffer*, if specified, and into the unnamed buffer.

If there are no characters before the current character on the current line, it shall be an error. If there are less than *count* previous characters on the current line, *count* shall be adjusted to the number of previous characters on the line.

Current line: Unchanged.

Current column: Set to (*current column – the width of the deleted characters*).

Yank

Synopsis: `[buffer][count] y motion`

Copy (yank) the region of text into *buffer*, if specified, and into the unnamed buffer.

If the motion command is the **y** command repeated:

1. The buffer shall be in line mode.
2. If there are less than *count* –1 lines after the current line in the edit buffer, it shall be an error.
3. The text region shall be from the current line up to and including the next *count* –1 lines.

Otherwise, the buffer text mode and text region shall be as specified by the motion command.

Current line: If the motion was from the current cursor position toward the end of the edit buffer, unchanged. Otherwise, set to the first line in the edit buffer that is part of the text region specified by the motion command.

Current column:

1. If the motion was from the current cursor position toward the end of the edit buffer, unchanged.
2. Otherwise, if the current line is empty, set to column position 1.
3. Otherwise, set to the last column that displays any part of the first character in the file that is part of the text region specified by the motion command.

Yank Current Line

Synopsis: `[buffer][count] Y`

This command shall be equivalent to the *vi* command:

```
[buffer][count] y_
```

Redraw Window

If in open mode, the **z** command shall have the Synopsis:

Synopsis: `[count] z`

If *count* is not specified, it shall default to the **window** edit option −1. The **z** command shall be equivalent to the *ex* **z** command, with a type character of **=** and a *count* of *count* −2, except that the current line and current column shall be set as follows, and the **window** edit option shall not be affected. If the calculation for the *count* argument would result in a negative number, the *count* argument to the *ex* **z** command shall be zero. A blank line shall be written after the last line is written.

Current line: Unchanged.

Current column: Unchanged.

If not in open mode, the **z** command shall have the following Synopsis:

Synopsis: `[line] z [count] character`

If *line* is not specified, it shall default to the current line. If *line* is specified, but is greater than the number of lines in the edit buffer, it shall default to the number of lines in the edit buffer.

If *count* is specified, the value of the **window** edit option shall be set to *count* (as described in the *ex* **window** command), and the screen shall be redrawn.

line shall be placed as specified by the following characters:

<newline>, <carriage-return>
Place the beginning of the line on the first line of the display.

. Place the beginning of the line in the center of the display. The middle line of the display shall be calculated as described for the **M** command.

− Place an unspecified portion of the line on the last line of the display.

\+ If *line* was specified, equivalent to the <newline> case. If *line* was not specified, display a screen where the first line of the display shall be (current last line) +1. If there are no lines after the last line in the display, it shall be an error.

^ If *line* was specified, display a screen where the last line of the display shall contain an unspecified portion of the first line of a display that had an unspecified portion of the specified line on the last line of the display. If this calculation results in a line before the beginning of the edit buffer, display the first screen of the edit buffer.

Otherwise, display a screen where the last line of the display shall contain an unspecified portion of (current first line −1). If this calculation results in a line before the beginning of the edit buffer, it shall be an error.

Current line: If *line* and the `'^'` character were specified:

1. If the first screen was displayed as a result of the command attempting to display lines before the beginning of the edit buffer: if the first screen was already displayed, unchanged; otherwise, set to (current first line −1).
2. Otherwise, set to the last line of the display.

If *line* and the `'+'` character were specified, set to the first line of the display.

Otherwise, if *line* was specified, set to *line*.

Otherwise, unchanged.

Current column: Set to non-<blank>.

Exit

Synopsis: `ZZ`

This command shall be equivalent to the *ex* **xit** command with no addresses, trailing **!**, or filename (see the *ex* **xit** command).

Input Mode Commands in vi

In text input mode, the current line shall consist of zero or more of the following categories, plus the terminating <newline>:

1. Characters preceding the text input entry point

   Characters in this category shall not be modified during text input mode.

2. **autoindent** characters

   autoindent characters shall be automatically inserted into each line that is created in text input mode, either as a result of entering a <newline> or <carriage-return> while in text input mode, or as an effect of the command itself; for example, **O** or **o** (see the *ex* **autoindent** command), as if entered by the user.

   It shall be possible to erase **autoindent** characters with the <control>-D command; it is unspecified whether they can be erased by <control>-H, <control>-U, and <control>-W characters. Erasing any **autoindent** character turns the glyph into erase-columns and deletes the character from the edit buffer, but does not change its representation on the screen.

3. Text input characters

   Text input characters are the characters entered by the user. Erasing any text input character turns the glyph into erase-columns and deletes the character from the edit buffer, but does not change its representation on the screen.

   Each text input character entered by the user (that does not have a special meaning) shall be treated as follows:

   a. The text input character shall be appended to the last character in the edit buffer from the first, second, or third categories.

   b. If there are no erase-columns on the screen, the text input command was the **R** command, and characters in the fifth category from the original line follow the cursor, the next such character shall be deleted from the edit buffer. If the **slowopen** edit option is not set, the corresponding glyph on the screen shall become erase-columns.

   c. If there are erase-columns on the screen, as many columns as they occupy, or as are necessary, shall be overwritten to display the text input character. (If only part of a multi-column glyph is overwritten, the remainder shall be left on the screen, and continue to be treated as erase-columns; it is unspecified whether the remainder of the glyph is modified in any way.)

   d. If additional display line columns are needed to display the text input character:

      1. If the **slowopen** edit option is set, the text input characters shall be displayed on subsequent display line columns, overwriting any characters displayed in

those columns.

2. Otherwise, any characters currently displayed on or after the column on the display line where the text input character is to be displayed shall be pushed ahead the number of display line columns necessary to display the rest of the text input character.

4. Erase-columns

   Erase-columns are not logically part of the edit buffer, appearing only on the screen, and may be overwritten on the screen by subsequent text input characters. When text input mode ends, all erase-columns shall no longer appear on the screen.

   Erase-columns are initially the region of text specified by the **c** command (see **Change** (on page 3207)) however, erasing **autoindent** or text input characters causes the glyphs of the erased characters to be treated as erase-columns.

5. Characters following the text region for the **c** command, or the text input entry point for all other commands

   Characters in this category shall not be modified during text input mode, except as specified in category 3.b. for the **R** text input command, or as <blank>s deleted when a <newline> or <carriage-return> is entered.

It is unspecified whether it is an error to attempt to erase past the beginning of a line that was created by the entry of a <newline> or <carriage-return> during text input mode. If it is not an error, the editor shall behave as if the erasing character was entered immediately after the last text input character entered on the previous line, and all of the non-<newline>s on the current line shall be treated as erase-columns.

When text input mode is entered, or after a text input mode character is entered (except as specified for the special characters below), the cursor shall be positioned as follows:

1. On the first column that displays any part of the first erase-column, if one exists
2. Otherwise, if the **slowopen** edit option is set, on the first display line column after the last character in the first, second, or third categories, if one exists
3. Otherwise, the first column that displays any part of the first character in the fifth category, if one exists
4. Otherwise, the display line column after the last character in the first, second, or third categories, if one exists
5. Otherwise, on column position 1

The characters that are updated on the screen during text input mode are unspecified, other than that the last text input character shall always be updated, and, if the **slowopen** edit option is not set, the current cursor character shall always be updated.

The following specifications are for command characters entered during text input mode.

NUL

Synopsis: `NUL`

If the first character of the text input is a NUL, the most recently input text shall be input as if entered by the user, and then text input mode shall be exited. The text shall be input literally; that is, characters are neither macro or abbreviation expanded, nor are any characters interpreted in any special manner. It is unspecified whether implementations shall support more than 256 bytes of remembered input text.

<control>-D

Synopsis: `<control>-D`

The <control>-D character shall have no special meaning when in text input mode for a line-oriented command (see **Command Descriptions in vi** (on page 3186)).

This command need not be supported on block-mode terminals.

If the cursor does not follow an **autoindent** character, or an **autoindent** character and a `'0'` or `'^'` character:

1. If the cursor is in column position 1, the <control>-D character shall be discarded and no further action taken.
2. Otherwise, the <control>-D character shall have no special meaning.

If the last input character was a `'0'`, the cursor shall be moved to column position 1.

Otherwise, if the last input character was a `'^'`, the cursor shall be moved to column position 1. In addition, the **autoindent** level for the next input line shall be derived from the same line from which the **autoindent** level for the current input line was derived.

Otherwise, the cursor shall be moved back to the column after the previous shiftwidth (see the *ex* **shiftwidth** command) boundary.

All of the glyphs on columns between the starting cursor position and (inclusively) the ending cursor position shall become erase-columns as described in **Input Mode Commands in vi** (on page 3220).

Current line: Unchanged.

Current column: Set to 1 if the <control>-D was preceded by a `'^'` or `'0'`; otherwise, set to (column −1) −((column −2) % **shiftwidth**).

<control>-H

Synopsis: `<control>-H`

If in text input mode for a line-oriented command, and there are no characters to erase, text input mode shall be terminated, no further action shall be done for this command, and the current line and column shall be unchanged.

If there are characters other than **autoindent** characters that have been input on the current line before the cursor, the cursor shall move back one character.

Otherwise, if there are **autoindent** characters on the current line before the cursor, it is implementation-defined whether the <control>-H command is an error or if the cursor moves back one **autoindent** character.

Otherwise, if the cursor is in column position 1 and there are previous lines that have been input, it is implementation-defined whether the <control>-H command is an error or if it is equivalent to entering <control>-H after the last input character on the previous input line.

Otherwise, it shall be an error.

All of the glyphs on columns between the starting cursor position and (inclusively) the ending cursor position shall become erase-columns as described in **Input Mode Commands in vi** (on page 3220).

The current erase character (see *stty*) shall cause an equivalent action to the <control>-H command, unless the previously inserted character was a backslash, in which case it shall be as if the literal current erase character had been inserted instead of the backslash.

Current line: Unchanged, unless previously input lines are erased, in which case it shall be set to line −1.

Current column: Set to the first column that displays any portion of the character backed up over.

<newline>

Synopsis:
```
<newline>
<carriage-return>
<control>-J
<control>-M
```

If input was part of a line-oriented command, text input mode shall be terminated and the command shall continue execution with the input provided.

Otherwise, terminate the current line. If there are no characters other than **autoindent** characters on the line, all characters on the line shall be discarded. Otherwise, it is unspecified whether the **autoindent** characters in the line are modified by entering these characters.

Continue text input mode on a new line appended after the current line. If the **slowopen** edit option is set, the lines on the screen below the current line shall not be pushed down, but the first of them shall be cleared and shall appear to be overwritten. Otherwise, the lines of the screen below the current line shall be pushed down.

If the **autoindent** edit option is set, an appropriate number of **autoindent** characters shall be added as a prefix to the line as described by the *ex* **autoindent** edit option.

All columns after the cursor that are erase-columns (as described in **Input Mode Commands in vi** (on page 3220)) shall be discarded.

If the **autoindent** edit option is set, all <blank>s immediately following the cursor shall be discarded.

All remaining characters after the cursor shall be transferred to the new line, positioned after any **autoindent** characters.

Current line: Set to current line +1.

Current column: Set to the first column that displays any portion of the first character after the **autoindent** characters on the new line, if any, or the first column position after the last **autoindent** character, if any, or column position 1.

<control>-T

Synopsis: `<control>-T`

The <control>-T character shall have no special meaning when in text input mode for a line-oriented command (see **Command Descriptions in vi** (on page 3186)).

This command need not be supported on block-mode terminals.

Behave as if the user entered the minimum number of <blank>s necessary to move the cursor forward to the column position after the next **shiftwidth** (see the *ex* **shiftwidth** command) boundary.

Current line: Unchanged.

Current column: Set to *column* + **shiftwidth** − ((column −1) % **shiftwidth**).

<control>-U

Synopsis: `<control>-U`

If there are characters other than **autoindent** characters that have been input on the current line before the cursor, the cursor shall move to the first character input after the **autoindent** characters.

Otherwise, if there are **autoindent** characters on the current line before the cursor, it is implementation-defined whether the <control>-U command is an error or if the cursor moves to the first column position on the line.

Otherwise, if the cursor is in column position 1 and there are previous lines that have been input, it is implementation-defined whether the <control>-U command is an error or if it is equivalent to entering <control>-U after the last input character on the previous input line.

Otherwise, it shall be an error.

All of the glyphs on columns between the starting cursor position and (inclusively) the ending cursor position shall become erase-columns as described in **Input Mode Commands in vi** (on page 3220).

The current *kill* character (see *stty*) shall cause an equivalent action to the <control>-U command, unless the previously inserted character was a backslash, in which case it shall be as if the literal current *kill* character had been inserted instead of the backslash.

Current line: Unchanged, unless previously input lines are erased, in which case it shall be set to line −1.

Current column: Set to the first column that displays any portion of the last character backed up over.

<control>-V

Synopsis:
```
<control>-V
<control>-Q
```

Allow the entry of any subsequent character, other than <control>-J or the <newline>, as a literal character, removing any special meaning that it may have to the editor in text input mode. If a <control>-V or <control>-Q is entered before a <control>-J or <newline>, the <control>-V or <control>-Q character shall be discarded, and the <control>-J or <newline> shall behave as described in the <newline> command character during input mode.

For purposes of the display only, the editor shall behave as if a `'^'` character was entered, and the cursor shall be positioned as if overwriting the `'^'` character. When a subsequent character is entered, the editor shall behave as if that character was entered instead of the original <control>-V or <control>-Q character.

Current line: Unchanged.

Current column: Unchanged.

<control>-W

Synopsis: `<control>-W`

If there are characters other than **autoindent** characters that have been input on the current line before the cursor, the cursor shall move back over the last word preceding the cursor (including any <blank>s between the end of the last word and the current cursor); the cursor shall not move to before the first character after the end of any **autoindent** characters.

Otherwise, if there are **autoindent** characters on the current line before the cursor, it is implementation-defined whether the <control>-W command is an error or if the cursor moves to the first column position on the line.

Otherwise, if the cursor is in column position 1 and there are previous lines that have been input, it is implementation-defined whether the <control>-W command is an error or if it is equivalent to entering <control>-W after the last input character on the previous input line.

Otherwise, it shall be an error.

All of the glyphs on columns between the starting cursor position and (inclusively) the ending cursor position shall become erase-columns as described in **Input Mode Commands in vi** (on page 3220).

Current line: Unchanged, unless previously input lines are erased, in which case it shall be set to line −1.

Current column: Set to the first column that displays any portion of the last character backed up over.

<ESC>

Synopsis: `<ESC>`

If input was part of a line-oriented command:

1. If *interrupt* was entered, text input mode shall be terminated and the editor shall return to command mode. The terminal shall be alerted.
2. If <ESC> was entered, text input mode shall be terminated and the command shall continue execution with the input provided.

Otherwise, terminate text input mode and return to command mode.

Any **autoindent** characters entered on newly created lines that have no other non-<newline>s shall be deleted.

Any leading **autoindent** and <blank>s on newly created lines shall be rewritten to be the minimum number of <blank>s possible.

The screen shall be redisplayed as necessary to match the contents of the edit buffer.

Current line: Unchanged.

Current column:

1. If there are text input characters on the current line, the column shall be set to the last column where any portion of the last text input character is displayed.
2. Otherwise, if a character is displayed in the current column, unchanged.
3. Otherwise, set to column position 1.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

When any error is encountered and the standard input is not a terminal device file, *vi* shall not write the file or return to command or text input mode, and shall terminate with a non-zero exit status.

Otherwise, when an unrecoverable error is encountered it shall be equivalent to a SIGHUP asynchronous event.

Otherwise, when an error is encountered, the editor shall behave as specified in **Command Descriptions in vi** (on page 3186).

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

See the RATIONALE for *ex* for more information on *vi*. Major portions of the *vi* utility specification point to *ex* to avoid inadvertent divergence. While *ex* and *vi* have historically been implemented as a single utility, this is not required by IEEE Std 1003.1-200x.

It is recognized that portions of *vi* would be difficult, if not impossible, to implement satisfactorily on a block-mode terminal, or a terminal without any form of cursor addressing, thus it is not a mandatory requirement that such features should work on all terminals. It is the intention, however, that a *vi* implementation should provide the full set of capabilities on all terminals capable of supporting them.

Historically, *vi* exited immediately if the standard input was not a terminal. IEEE Std 1003.1-200x permits, but does not require, this behavior. An end-of-file condition is not equivalent to an end-of-file character. A common end-of-file character, <control>-D, is historically a *vi* command.

The text in the STANDARD OUTPUT section reflects the usage of the verb *display* in this section; some implementations of *vi* use standard output to write to the terminal, but IEEE Std 1003.1-200x does not require that to be the case.

Historically, implementations reverted to open mode if the terminal was incapable of supporting full visual mode. IEEE Std 1003.1-200x requires this behavior. Historically, the open mode of *vi* behaved roughly equivalently to the visual mode, with the exception that only a single line from the edit buffer (one ‘‘buffer line’’) was kept current at any time. This line was normally displayed on the next-to-last line of a terminal with cursor addressing (and the last line performed its normal visual functions for line-oriented commands and messages). In addition, some few commands behaved differently in open mode than in visual mode. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, *ex* and *vi* implementations have expected text to proceed in the usual European/Latin order of left to right, top to bottom. There is no requirement in IEEE Std 1003.1-200x that this be the case. The specification was deliberately written using words like ''before'', ''after'', ''first'', and ''last'' in order to permit implementations to support the natural text order of the language.

Historically, lines past the end of the edit buffer were marked with single tilde (`'~'`) characters; that is, if the one-based display was 20 lines in length, and the last line of the file was on line one, then lines 2-20 would contain only a single `'~'` character.

Historically, the *vi* editor attempted to display only complete lines at the bottom of the screen (it did display partial lines at the top of the screen). If a line was too long to fit in its entirety at the bottom of the screen, the screen lines where the line would have been displayed were displayed as single `'@'` characters, instead of displaying part of the line. IEEE Std 1003.1-200x permits, but does not require, this behavior. Implementations are encouraged to attempt always to display a complete line at the bottom of the screen when doing scrolling or screen positioning by buffer lines.

Historically, lines marked with `'@'` were also used to minimize output to dumb terminals over slow lines; that is, changes local to the cursor were updated, but changes to lines on the screen that were not close to the cursor were simply marked with an `'@'` sign instead of being updated to match the current text. IEEE Std 1003.1-200x permits, but does not require this feature because it is used ever less frequently as terminals become smarter and connections are faster.

Initialization in ex and vi

Historically, *vi* always had a line in the edit buffer, even if the edit buffer was ''empty''. For example:

1. The *ex* command **=** executed from visual mode wrote ''1'' when the buffer was empty.
2. Writes from visual mode of an empty edit buffer wrote files of a single character (a <newline>), while writes from *ex* mode of an empty edit buffer wrote empty files.
3. Put and read commands into an empty edit buffer left an empty line at the top of the edit buffer.

For consistency, IEEE Std 1003.1-200x does not permit any of these behaviors.

Historically, *vi* did not always return the terminal to its original modes; for example, ICRNL was modified if it was not originally set. IEEE Std 1003.1-200x does not permit this behavior.

Command Descriptions in vi

Motion commands are among the most complicated aspects of *vi* to describe. With some exceptions, the text region and buffer type effect of a motion command on a *vi* command are described on a case-by-case basis. The descriptions of text regions in IEEE Std 1003.1-200x are not intended to imply direction; that is, an inclusive region from line n to line $n+5$ is identical to a region from line $n+5$ to line n. This is of more than academic interest—movements to marks can be in either direction, and, if the **wrapscan** option is set, so can movements to search points. Historically, lines are always stored into buffers in text order; that is, from the start of the edit buffer to the end. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, command counts were applied to any associated motion, and were multiplicative to any supplied motion count. For example, **2cw** is the same as **c2w**, and **2c3w** is the same as **c6w**. IEEE Std 1003.1-200x requires this behavior. Historically, *vi* commands that used bigwords, words, paragraphs, and sentences as objects treated groups of empty lines, or lines that contained only <blank>s, inconsistently. Some commands treated them as a single entity, while others treated each line separately. For example, the **w**, **W**, and **B** commands treated groups of empty lines as individual words; that is, the command would move the cursor to each new empty line. The **e** and **E** commands treated groups of empty lines as a single word; that is, the first use would move past the group of lines. The **b** command would just beep at the user, or if done from the start of the line as a motion command, fail in unexpected ways. If the lines contained only (or ended with) <blank>s, the **w** and **W** commands would just beep at the user, the **E** and **e** commands would treat the group as a single word, and the **B** and **b** commands would treat the lines as individual words. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that all *vi* commands treat groups of empty or blank lines as a single entity, and that movement through lines ending with <blank>s be consistent with other movements.

Historically, *vi* documentation indicated that any number of double quotes were skipped after punctuation marks at sentence boundaries; however, implementations only skipped single quotes. IEEE Std 1003.1-200x requires both to be skipped.

Historically, the first and last characters in the edit buffer were word boundaries. This historical practice is required by IEEE Std 1003.1-200x.

Historically, *vi* attempted to update the minimum number of columns on the screen possible, which could lead to misleading information being displayed. IEEE Std 1003.1-200x makes no requirements other than that the current character being entered is displayed correctly, leaving all other decisions in this area up to the implementation.

Historically, lines were arbitrarily folded between columns of any characters that required multiple column positions on the screen, with the exception of tabs, which terminated at the right-hand margin. IEEE Std 1003.1-200x permits the former and requires the latter. Implementations that do not arbitrarily break lines between columns of characters that occupy multiple column positions should not permit the cursor to rest on a column that does not contain any part of a character.

The historical *vi* had a problem in that all movements were by buffer lines, not by display or screen lines. This is often the right thing to do; for example, single line movements, such as **j** or **k**, should work on buffer lines. Commands like **dj**, or **j.**, where **.** is a change command, only make sense for buffer lines. It is not, however, the right thing to do for screen motion or scrolling commands like <control>-D, <control>-F, and **H**. If the window is fairly small, using buffer lines in these cases can result in completely random motion; for example, **1**<control>-D can result in a completely changed screen, without any overlap. This is clearly not what the user wanted. The problem is even worse in the case of the **H**, **L**, and **M** commands—as they position the cursor at the first non-<blank> of the line, they may all refer to the same location in large lines, and will result in no movement at all.

In addition, if the line is larger than the screen, using buffer lines can make it impossible to display parts of the line—there are not any commands that do not display the beginning of the line in historical *vi*, and if both the beginning and end of the line cannot be on the screen at the same time, the user suffers. Finally, the page and half-page scrolling commands historically moved to the first non-<blank> in the new line. If the line is approximately the same size as the screen, this is inadequate because the cursor before and after a <control>-D command will refer to the same location on the screen.

Implementations of *ex* and *vi* exist that do not have these problems because the relevant commands (<control>-B, <control>-D, <control>-F, <control>-U, <control>-Y, <control>-E, **H**, **L**, and **M)** operate on display (screen) lines, not (edit) buffer lines.

IEEE Std 1003.1-200x does not permit this behavior by default because the standard developers believed that users would find it too confusing. However, historical practice has been relaxed.

For example, *ex* and *vi* historically attempted, albeit sometimes unsuccessfully, to never put part of a line on the last lines of a screen; for example, if a line would not fit in its entirety, no part of the line was displayed, and the screen lines corresponding to the line contained single '@' characters. This behavior is permitted, but not required by IEEE Std 1003.1-200x, so that it is possible for implementations to support long lines in small screens more reasonably without changing the commands to be oriented to the display (instead of oriented to the buffer). IEEE Std 1003.1-200x also permits implementations to refuse to edit any edit buffer containing a line that will not fit on the screen in its entirety.

The display area (for example, the value of the **window** edit option) has historically been ‘‘grown’’, or expanded, to display new text when local movements are done in displays where the number of lines displayed is less than the maximum possible. Expansion has historically been the first choice, when the target line is less than the maximum possible expansion value away. Scrolling has historically been the next choice, done when the target line is less than half a display away, and otherwise, the screen was redrawn. There were exceptions, however, in that *ex* commands generally always caused the screen to be redrawn. IEEE Std 1003.1-200x does not specify a standard behavior because there may be external issues, such as connection speed, the number of characters necessary to redraw as opposed to scroll, or terminal capabilities that implementations will have to accommodate.

The current line in IEEE Std 1003.1-200x maps one-to-one to a buffer line in the file. The current column does not. There are two different column values that are described by IEEE Std 1003.1-200x. The first is the current column value as set by many of the *vi* commands. This value is remembered for the lifetime of the editor. The second column value is the actual position on the screen where the cursor rests. The two are not always the same. For example, when the cursor is backed by a multi-column character, the actual cursor position on the screen has historically been the last column of the character in command mode, and the first column of the character in input mode.

Commands that set the current line, but that do not set the current cursor value (for example, **j** and **k**) attempt to get as close as possible to the remembered column position, so that the cursor tends to restrict itself to a vertical column as the user moves around in the edit buffer. IEEE Std 1003.1-200x requires conformance to historical practice, requiring that the display location of the cursor on the display line be adjusted from the current column value as necessary to support this historical behavior.

Historically, only a single line (and for some terminals, a single line minus 1 column) of characters could be entered by the user for the line oriented commands; that is, **:**, **!**, **/**, or **?**. IEEE Std 1003.1-200x permits, but does not require, this limitation.

Historically, ‘‘soft’’ errors in *vi* caused the terminal to be alerted, but no error message was displayed. As a general rule, no error message was displayed for errors in command execution in *vi*, when the error resulted from the user attempting an invalid or impossible action, or when a searched-for object was not found. Examples of soft errors included **h** at the left margin, <control>-B or **[[** at the beginning of the file, **2G** at the end of the file, and so on. In addition, errors such as **%**, **]]**, **}**, **)**, **N**, **n**, **f**, **F**, **t**, and **T** failing to find the searched-for object were soft as well. Less consistently, **/** and **?** displayed an error message if the pattern was not found, **/**, **?**, **N**, and **n** displayed an error message if no previous regular expression had been specified, and **;** did not display an error message if no previous **f**, **F**, **t**, or **T** command had occurred. Also, behavior in this area might reasonably be based on a runtime evaluation of the speed of a network connection. Finally, some implementations have provided error messages for soft errors in order to assist naive users, based on the value of a verbose edit option. IEEE Std 1003.1-200x does not list specific errors for which an error message shall be displayed. Implementations should conform to historical practice in the absence of any strong reason to diverge.

Page Backwards

The <control>-B and <control>-F commands historically considered it an error to attempt to page past the beginning or end of the file, whereas the <control>-D and <control>-U commands simply moved to the beginning or end of the file. For consistency, IEEE Std 1003.1-200x requires the latter behavior for all four commands. All four commands still consider it an error if the current line is at the beginning (<control>-B, <control>-U) or end (<control>-F, <control>-D) of the file. Historically, the <control>-B and <control>-F commands skip two lines in order to include overlapping lines when a single command is entered. This makes less sense in the presence of a *count*, as there will be, by definition, no overlapping lines. The actual calculation used by historical implementations of the *vi* editor for <control>-B was:

```
((current first line) − count x (window edit option)) +2
```

and for <control>-F was:

```
((current first line) + count x (window edit option)) −2
```

This calculation does not work well when intermixing commands with and without counts; for example, **3**<control>-F is not equivalent to entering the <control>-F command three times, and is not reversible by entering the <control>-B command three times. For consistency with other *vi* commands that take counts, IEEE Std 1003.1-200x requires a different calculation.

Scroll Forward

The 4BSD and System V implementations of *vi* differed on the initial value used by the **scroll** command. 4BSD used:

```
((window edit option) +1) /2
```

while System V used the value of the **scroll** edit option. The System V version is specified by IEEE Std 1003.1-200x because the standard developers believed that it was more intuitive and permitted the user a method of setting the scroll value initially without also setting the number of lines that are displayed.

Scroll Forward by Line

Historically, the <control>-E and <control>-Y commands considered it an error if the last and first lines, respectively, were already on the screen. IEEE Std 1003.1-200x requires conformance to historical practice. Historically, the <control>-E and <control>-Y commands had no effect in open mode. For simplicity and consistency of specification, IEEE Std 1003.1-200x requires that they behave as usual, albeit with a single line screen.

Clear and Redisplay

The historical <control>-L command refreshed the screen exactly as it was supposed to be currently displayed, replacing any `'@'` characters for lines that had been deleted but not updated on the screen with refreshed `'@'` characters. The intent of the <control>-L command is to refresh when the screen has been accidentally overwritten; for example, by a **write** command from another user, or modem noise.

Redraw Screen

The historical <control>-R command redisplayed only when necessary to update lines that had been deleted but not updated on the screen and that were flagged with `'@'` characters. There is no requirement that the screen be in any way refreshed if no lines of this form are currently displayed. IEEE Std 1003.1-200x permits implementations to extend this command to refresh lines on the screen flagged with `'@'` characters because they are too long to be displayed in the current framework; however, the current line and column need not be modified.

Search for tagstring

Historically, the first non-<blank> at or after the cursor was the first character, and all subsequent characters that were word characters, up to the end of the line, were included. For example, with the cursor on the leading space or on the `'#'` character in the text `"#bar@"`, the tag was `"#bar"`. On the character `'b'` it was `"bar"`, and on the `'a'`, it was `"ar"`. IEEE Std 1003.1-200x requires this behavior.

Replace Text with Results from Shell Command

Historically, the **<**, **>**, and **!** commands considered most cursor motions other than line-oriented motions an error; for example, the command **>/foo<CR>** succeeded, while the command **>l** failed, even though the text region described by the two commands might be identical. For consistency, all three commands only consider entire lines and not partial lines, and the region is defined as any line that contains a character that was specified by the motion.

Move to Matching Character

Other matching characters have been left implementation-defined in order to allow extensions such as matching `'<'` and `'>'` for searching HTML, or **#ifdef**, **#else**, and **#endif** for searching C source.

Repeat Substitution

IEEE Std 1003.1-200x requires that any **c** and **g** flags specified to the previous substitute command be ignored; however, the **r** flag may still apply, if supported by the implementation.

Return to Previous (Context or Section)

The **[[**, **]]**, **(**, **)**, **{**, and **}** commands are all affected by ‘‘section boundaries’’, but in some historical implementations not all of the commands recognize the same section boundaries. This is a bug, not a feature, and a unique section-boundary algorithm was not described for each command. One special case that is preserved is that the sentence command moves to the end of the last line of the edit buffer while the other commands go to the beginning, in order to preserve the traditional character cut semantics of the sentence command. Historically, *vi* section boundaries at the beginning and end of the edit buffer were the first non-<blank> on the first and last lines of the edit buffer if one exists; otherwise, the last character of the first and last lines of the edit buffer if one exists. To increase consistency with other section locations, this has been simplified by IEEE Std 1003.1-200x to the first character of the first and last lines of the edit buffer, or the first and the last lines of the edit buffer if they are empty.

Sentence boundaries were problematic in the historical *vi*. They were not only the boundaries as defined for the section and paragraph commands, but they were the first non-<blank> that occurred after those boundaries, as well. Historically, the *vi* section commands were documented as taking an optional window size as a *count* preceding the command. This was not implemented in historical versions, so IEEE Std 1003.1-200x requires that the *count* repeat the command, for consistency with other *vi* commands.

Repeat

Historically, mapped commands other than text input commands could not be repeated using the **period** command. IEEE Std 1003.1-200x requires conformance to historical practice.

The restrictions on the interpretation of special characters (for example, <control>-H) in the repetition of text input mode commands is intended to match historical practice. For example, given the input sequence:

```
iab<control>-H<control>-H<control>-Hdef<escape>
```

the user should be informed of an error when the sequence is first entered, but not during a command repetition. The character <control>-T is specifically exempted from this restriction. Historical implementations of *vi* ignored <control>-T characters that were input in the original command during command repetition. IEEE Std 1003.1-200x prohibits this behavior.

Find Regular Expression

Historically, commands did not affect the line searched to or from if the motion command was a search (**/**, **?**, **N**, **n**) and the final position was the start/end of the line. There were some special cases and *vi* was not consistent. IEEE Std 1003.1-200x does not permit this behavior, for consistency. Historical implementations permitted, but were unable to handle searches as motion commands that wrapped (that is, due to the edit option **wrapscan**) to the original location. IEEE Std 1003.1-200x requires that this behavior be treated as an error.

Historically, the syntax `"/RE/0"` was used to force the command to cut text in line mode. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, in open mode, a **z** specified to a search command redisplayed the current line instead of displaying the current screen with the current line highlighted. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Historically, trailing **z** commands were permitted and ignored if entered as part of a search used as a motion command. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Execute an ex Command

Historically, *vi* implementations restricted the commands that could be entered on the colon command line (for example, **append** and **change**), and some other commands were known to cause them to fail catastrophically. For consistency, IEEE Std 1003.1-200x does not permit these restrictions. When executing an *ex* command by entering **:**, it is not possible to enter a <newline> as part of the command because it is considered the end of the command. A different approach is to enter *ex* command mode by using the *vi* **Q** command (and later resuming visual mode with the *ex* **vi** command). In *ex* command mode, the single-line limitation does not exist. So, for example, the following is valid:

```
Q
s/break here/break\
here/
vi
```

IEEE Std 1003.1-200x requires that, if the *ex* command overwrites any part of the screen that would be erased by a refresh, *vi* pauses for a character from the user. Historically, this character could be any character; for example, a character input by the user before the message appeared, or even a mapped character. This is probably a bug, but implementations that have tried to be more rigorous by requiring that the user enter a specific character, or that the user enter a character after the message was displayed, have been forced by user indignation back into historical behavior. IEEE Std 1003.1-200x requires conformance to historical practice.

Shift Left (Right)

Refer to the Rationale for the **!** and **/** commands. Historically, the **<** and **>** commands sometimes moved the cursor to the first non-<blank> (for example if the command was repeated or with **_** as the motion command), and sometimes left it unchanged. IEEE Std 1003.1-200x does not permit this inconsistency, requiring instead that the cursor always move to the first non-<blank>. Historically, the **<** and **>** commands did not support buffer arguments, although some implementations allow the specification of an optional buffer. This behavior is neither required nor disallowed by IEEE Std 1003.1-200x.

Execute

Historically, buffers could execute other buffers, and loops, infinite and otherwise, were possible. IEEE Std 1003.1-200x requires conformance to historical practice. The **buffer* syntax of *ex* is not required in *vi*, because it is not historical practice and has been used in some *vi* implementations to support additional scripting languages.

Reverse Case

Historically, the **~** command ignored any associated *count*, and acted only on the characters in the current line. For consistency with other *vi* commands, IEEE Std 1003.1-200x requires that an associated *count* act on the next *count* characters, and that the command move to subsequent lines if warranted by *count*, to make it possible to modify large pieces of text in a reasonably efficient manner. There exist *vi* implementations that optionally require an associated motion command for the **~** command. Implementations supporting this functionality are encouraged to base it on the **tildedop** edit option and handle the text regions and cursor positioning identically to the **yank** command.

Append

Historically, *count*s specified to the **A**, **a**, **I**, and **i** commands repeated the input of the first line *count* times, and did not repeat the subsequent lines of the input text. IEEE Std 1003.1-200x requires that the entire text input be repeated *count* times.

Move Backward to Preceding Word

Historically, *vi* became confused if word commands were used as motion commands in empty files. IEEE Std 1003.1-200x requires that this be an error. Historical implementations of *vi* had a large number of bugs in the word movement commands, and they varied greatly in behavior in the presence of empty lines, ‘‘words’’ made up of a single character, and lines containing only <blank>s. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Change to End-of-Line

Some historical implementations of the **C** command did not behave as described by IEEE Std 1003.1-200x when the **$** key was remapped because they were implemented by pushing the **$** key onto the input queue and reprocessing it. IEEE Std 1003.1-200x does not permit this behavior. Historically, the **C**, **S**, and **s** commands did not copy replaced text into the numeric buffers. For consistency and simplicity of specification, IEEE Std 1003.1-200x requires that they behave like their respective **c** commands in all respects.

Delete

Historically, lines in open mode that were deleted were scrolled up, and an **@** glyph written over the beginning of the line. In the case of terminals that are incapable of the necessary cursor motions, the editor erased the deleted line from the screen. IEEE Std 1003.1-200x requires conformance to historical practice; that is, if the terminal cannot display the '@' character, the line cannot remain on the screen.

Delete to End-of-Line

Some historical implementations of the **D** command did not behave as described by IEEE Std 1003.1-200x when the **$** key was remapped because they were implemented by pushing the **$** key onto the input queue and reprocessing it. IEEE Std 1003.1-200x does not permit this behavior.

Join

An historical oddity of *vi* is that the commands **J**, **1J**, and **2J** are all equivalent. IEEE Std 1003.1-200x requires conformance to historical practice. The *vi* **J** command is specified in terms of the *ex* **join** command with an *ex* command *count* value. The address correction for a *count* that is past the end of the edit buffer is necessary for historical compatibility for both *ex* and *vi*.

Mark Position

Historical practice is that only lowercase letters, plus '‘' and ''', could be used to mark a cursor position. IEEE Std 1003.1-200x requires conformance to historical practice, but encourages implementations to support other characters as marks as well.

Repeat Regular Expression Find (Forward and Reverse)

Historically, the **N** and **n** commands could not be used as motion components for the **c** command. With the exception of the **cN** command, which worked if the search crossed a line boundary, the text region would be discarded, and the user would not be in text input mode. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Insert Empty Line (Below and Above)

Historically, counts to the **O** and **o** commands were used as the number of physical lines to open, if the terminal was dumb and the **slowopen** option was not set. This was intended to minimize traffic over slow connections and repainting for dumb terminals. IEEE Std 1003.1-200x does not permit this behavior, requiring that a *count* to the open command behave as for other text input commands. This change to historical practice was made for consistency, and because a superset of the functionality is provided by the **slowopen** edit option.

Put from Buffer (Following and Before)

Historically, *count*s to the **p** and **P** commands were ignored if the buffer was a line mode buffer, but were (mostly) implemented as described in IEEE Std 1003.1-200x if the buffer was a character mode buffer. Because implementations exist that do not have this limitation, and because pasting lines multiple times is generally useful, IEEE Std 1003.1-200x requires that *count* be supported for all **p** and **P** commands.

Historical implementations of *vi* were widely known to have major problems in the **p** and **P** commands, particularly when unusual regions of text were copied into the edit buffer. The standard developers viewed these as bugs, and they are not permitted for consistency and simplicity of specification.

Historically, a **P** or **p** command (or an *ex* **put** command executed from open or visual mode) executed in an empty file, left an empty line as the first line of the file. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Replace Character

Historically, the **r** command did not correctly handle the *erase* and *word erase* characters as arguments, nor did it handle an associated *count* greater than 1 with a <carriage-return> argument, for which it replaced *count* characters with a single <newline>. IEEE Std 1003.1-200x does not permit these inconsistencies.

Historically, the **r** command permitted the <control>-V escaping of entered characters, such as <ESC> and the <carriage-return>; however, it required two leading <control>-V characters instead of one. IEEE Std 1003.1-200x requires that this be changed for consistency with the other text input commands of *vi*.

Historically, it is an error to enter the **r** command if there are less than *count* characters at or after the cursor in the line. While a reasonable and unambiguous extension would be to permit the **r** command on empty lines, it would require that too large a *count* be adjusted to match the number of characters at or after the cursor for consistency, which is sufficiently different from historical practice to be avoided. IEEE Std 1003.1-200x requires conformance to historical practice.

Replace Characters

Historically, if there were **autoindent** characters in the line on which the **R** command was run, and **autoindent** was set, the first <newline> would be properly indented and no characters would be replaced by the <newline>. Each additional <newline> would replace *n* characters, where *n* was the number of characters that were needed to indent the rest of the line to the proper indentation level. This behavior is a bug and is not permitted by IEEE Std 1003.1-200x.

Undo

Historical practice for cursor positioning after undoing commands was mixed. In most cases, when undoing commands that affected a single line, the cursor was moved to the start of added or changed text, or immediately after deleted text. However, if the user had moved from the line being changed, the column was either set to the first non-<blank>, returned to the origin of the command, or remained unchanged. When undoing commands that affected multiple lines or entire lines, the cursor was moved to the first character in the first line restored. As an example of how inconsistent this was, a search, followed by an **o** text input command, followed by an **undo** would return the cursor to the location where the **o** command was entered, but a **cw** command followed by an **o** command followed by an **undo** would return the cursor to the first non-<blank> of the line. IEEE Std 1003.1-200x requires the most useful of these behaviors, and discards the least useful, in the interest of consistency and simplicity of specification.

Yank

Historically, the **yank** command did not move to the end of the motion if the motion was in the forward direction. It moved to the end of the motion if the motion was in the backward direction, except for the **_** command, or for the **G** and **'** commands when the end of the motion was on the current line. This was further complicated by the fact that for a number of motion commands, the **yank** command moved the cursor but did not update the screen; for example, a subsequent command would move the cursor from the end of the motion, even though the cursor on the screen had not reflected the cursor movement for the **yank** command. IEEE Std 1003.1-200x requires that all **yank** commands associated with backward motions move the cursor to the end of the motion for consistency, and specifically, to make **'** commands as motions consistent with search patterns as motions.

Yank Current Line

Some historical implementations of the **Y** command did not behave as described by IEEE Std 1003.1-200x when the '_' key was remapped because they were implemented by pushing the '_' key onto the input queue and reprocessing it. IEEE Std 1003.1-200x does not permit this behavior.

Redraw Window

Historically, the **z** command always redrew the screen. This is permitted but not required by IEEE Std 1003.1-200x, because of the frequent use of the **z** command in macros such as **map n nz.** for screen positioning, instead of its use to change the screen size. The standard developers believed that expanding or scrolling the screen offered a better interface for users. The ability to redraw the screen is preserved if the optional new window size is specified, and in the <control>-L and <control>-R commands.

The semantics of **z**^ are confusing at best. Historical practice is that the screen before the screen that ended with the specified line is displayed. IEEE Std 1003.1-200x requires conformance to historical practice.

Historically, the **z** command would not display a partial line at the top or bottom of the screen. If the partial line would normally have been displayed at the bottom of the screen, the command worked, but the partial line was replaced with '@' characters. If the partial line would normally have been displayed at the top of the screen, the command would fail. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Historically, the **z** command with a line specification of 1 ignored the command. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Historically, the **z** command did not set the cursor column to the first non-<blank> for the character if the first screen was to be displayed, and was already displayed. For consistency and simplicity of specification, IEEE Std 1003.1-200x does not permit this behavior.

Input Mode Commands in vi

Historical implementations of *vi* did not permit the user to erase more than a single line of input, or to use normal erase characters such as *line erase*, *worderase*, and *erase* to erase **autoindent** characters. As there exist implementations of *vi* that do not have these limitations, both behaviors are permitted, but only historical practice is required. In the case of these extensions, *vi* is required to pause at the **autoindent** and previous line boundaries.

Historical implementations of *vi* updated only the portion of the screen where the current cursor character was displayed. For example, consider the *vi* input keystrokes:

```
iabcd<escape>0C<tab>
```

Historically, the <tab> would overwrite the characters `"abcd"` when it was displayed. Other implementations replace only the `'a'` character with the <tab>, and then push the rest of the characters ahead of the cursor. Both implementations have problems. The historical implementation is probably visually nicer for the above example; however, for the keystrokes:

```
iabcd<ESC>0R<tab><ESC>
```

the historical implementation results in the string `"bcd"` disappearing and then magically reappearing when the <ESC> character is entered. IEEE Std 1003.1-200x requires the former behavior when overwriting erase-columns; that is, overwriting characters that are no longer logically part of the edit buffer, and the latter behavior otherwise.

Historical implementations of *vi* discarded the <control>-D and <control>-T characters when they were entered at places where their command functionality was not appropriate. IEEE Std 1003.1-200x requires that the <control>-T functionality always be available, and that <control>-D be treated as any other key when not operating on **autoindent** characters.

NUL

Some historical implementations of *vi* limited the number of characters entered using the NUL input character to 256 bytes. IEEE Std 1003.1-200x permits this limitation; however, implementations are encouraged to remove this limit.

<control>-D

See also Rationale for the input mode command <newline>. The hidden assumptions in the <control>-D command (and in the *vi* **autoindent** specification in general) is that <space>s take up a single column on the screen and that <tab>s are comprised of an integral number of <space>s.

<newline>

Implementations are permitted to rewrite **autoindent** characters in the line when <newline>, <carriage-return>, <control>-D, and <control>-T are entered, or when the **shift** commands are used, because historical implementations have both done so and found it necessary to do so. For example, a <control>-D when the cursor is preceded by a single <tab>, with **tabstop** set to 8, and **shiftwidth** set to 3, will result in the <tab> being replaced by several <space>s.

<control>-T

See also the Rationale for the input mode command <newline>. Historically, <control>-T only worked if no non-<blank>s had yet been input in the current input line. In addition, the characters inserted by <control>-T were treated as **autoindent** characters, and could not be erased using normal user erase characters. Because implementations exist that do not have these limitations, and as moving to a column boundary is generally useful, IEEE Std 1003.1-200x requires that both limitations be removed.

<control>-V

Historically, *vi* used **ˆV**, regardless of the value of the literal-next character of the terminal. IEEE Std 1003.1-200x requires conformance to historical practice.

The uses described for <control>-V can also be accomplished with <control>-Q, which is useful on terminals that use <control>-V for the down-arrow function. However, most historical implementations use <control>-Q for the *termios* START character, so the editor will generally not receive the <control>-Q unless **stty ixon** mode is set to off. (In addition, some historical implementations of *vi* explicitly set **ixon** mode to on, so it was difficult for the user to set it to off.) Any of the command characters described in IEEE Std 1003.1-200x can be made ineffective by their selection as *termios* control characters, using the *stty* utility or other methods described in the System Interfaces volume of IEEE Std 1003.1-200x.

<ESC>

Historically, SIGINT alerted the terminal when used to end input mode. This behavior is permitted, but not required, by IEEE Std 1003.1-200x.

FUTURE DIRECTIONS

None.

SEE ALSO

ex, *stty*

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The APPLICATION USAGE section is added.

The obsolescent SYNOPSIS is removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The **reindent** command description is added.

The *vi* utility has been extensively rewritten for alignment with the IEEE P1003.2b draft standard.

IEEE PASC Interpretations 1003.2 #57, #62, #63, #64, #78, and #188 are applied. |

IEEE PASC Interpretation 1003.2 #207 is applied, clarifying the description of the **R** command in a manner similar to the descriptions of other text input mode commands such as **i**, **o**, and **O**. |

NAME

wait — await process completion

SYNOPSIS

```
wait [pid...]
```

DESCRIPTION

When an asynchronous list (see Section 2.9.3.1 (on page 2252)) is started by the shell, the process ID of the last command in each element of the asynchronous list shall become known in the current shell execution environment; see Section 2.12 (on page 2263).

If the *wait* utility is invoked with no operands, it shall wait until all process IDs known to the invoking shell have terminated and exit with a zero exit status.

If one or more *pid* operands are specified that represent known process IDs, the *wait* utility shall wait until all of them have terminated. If one or more *pid* operands are specified that represent unknown process IDs, *wait* shall treat them as if they were known process IDs that exited with exit status 127. The exit status returned by the *wait* utility shall be the exit status of the process requested by the last *pid* operand.

The known process IDs are applicable only for invocations of *wait* in the current shell execution environment.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

pid One of the following:

1. The unsigned decimal integer process ID of a command, for which the utility is to wait for the termination.
2. A job control job ID (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.203, Job Control Job ID) that identifies a background process group to be waited for. The job control job ID notation is applicable only for invocations of *wait* in the current shell execution environment; see Section 2.12 (on page 2263). The exit status of *wait* shall be determined by the last command in the pipeline.

   Note: The job control job ID type of *pid* is only available on systems supporting the User Portability Utilities option.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *wait*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If one or more operands were specified, all of them have terminated or were not known by the invoking shell, and the status of the last operand specified is known, then the exit status of *wait* shall be the exit status information of the command indicated by the last operand specified. If the process terminated abnormally due to the receipt of a signal, the exit status shall be greater than 128 and shall be distinct from the exit status generated by other signals, but the exact value is unspecified. (See the *kill* **−l** option.) Otherwise, the *wait* utility shall exit with one of the following values:

0	The *wait* utility was invoked with no operands and all process IDs known by the invoking shell have terminated.
1-126	The *wait* utility detected an error.
127	The command identified by the last *pid* operand specified is unknown.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

On most implementations, *wait* is a shell built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(wait)
nohup wait ...
find . −exec wait ... \;
```

it returns immediately because there are no known process IDs to wait for in those environments.

Historical implementations of interactive shells have discarded the exit status of terminated background processes before each shell prompt. Therefore, the status of background processes was usually lost unless it terminated while *wait* was waiting for it. This could be a serious problem when a job that was expected to run for a long time actually terminated quickly with a syntax or initialization error because the exit status returned was usually zero if the requested process ID was not found. This volume of IEEE Std 1003.1-200x requires the implementation to keep the status of terminated jobs available until the status is requested, so that scripts like:

```
j1&
p1=$!
j2&
wait $p1
echo Job 1 exited with status $?
wait $!
echo Job 2 exited with status $?
```

works without losing status on any of the jobs. The shell is allowed to discard the status of any process that it determines the application cannot get the process ID from the shell. It is also required to remember only {CHILD_MAX} number of processes in this way. Since the only way to get the process ID from the shell is by using the `'!'` shell parameter, the shell is allowed to discard the status of an asynchronous list if `"$!"` was not referenced before another asynchronous list was started. (This means that the shell only has to keep the status of the last asynchronous list started if the application did not reference `"$!"`. If the implementation of the shell is smart enough to determine that a reference to `"$!"` was not saved anywhere that the application can retrieve it later, it can use this information to trim the list of saved information. Note also that a successful call to *wait* with no operands discards the exit status of all asynchronous lists.)

If the exit status of *wait* is greater than 128, there is no way for the application to know if the waited-for process exited with that value or was killed by a signal. Since most utilities exit with small values, there is seldom any ambiguity. Even in the ambiguous cases, most applications just need to know that the asynchronous job failed; it does not matter whether it detected an error and failed or was killed and did not complete its job normally.

EXAMPLES

Although the exact value used when a process is terminated by a signal is unspecified, if it is known that a signal terminated a process, a script can still reliably figure out which signal using *kill* as shown by the following script:

```
sleep 1000&
pid=$!
kill -kill $pid
wait $pid
echo $pid was terminated by a SIG$(kill -l $?) signal.
```

If the following sequence of commands is run in less than 31 seconds:

```
sleep 257 | sleep 31 &
jobs -l %%
```

either of the following commands returns the exit status of the second *sleep* in the pipeline:

```
wait <pid of sleep 31>
wait %%
```

RATIONALE

The description of *wait* does not refer to the *waitpid*( ) function from the System Interfaces volume of IEEE Std 1003.1-200x because that would needlessly overspecify this interface. However, the wording means that *wait* is required to wait for an explicit process when it is given an argument so that the status information of other processes is not consumed. Historical implementations use the *wait*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x until *wait*( ) returns the requested process ID or finds that the requested process does not exist. Because this means that a shell script could not reliably get the status of all background children if a second background job was ever started before the first job finished, it is recommended that the *wait* utility use a method such as the functionality provided by the *waitpid*( ) function.

The ability to wait for multiple *pid* operands was adopted from the KornShell.

This new functionality was added because it is needed to determine the exit status of any asynchronous list accurately. The only compatibility problem that this change creates is for a script like

```
while sleep 60 do
    job& echo Job started $(date) as $!  done
```

which causes the shell to monitor all of the jobs started until the script terminates or runs out of memory. This would not be a problem if the loop did not reference `"$!"` or if the script would occasionally *wait* for jobs it started.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the System Interfaces volume of IEEE Std 1003.1-200x, *waitpid*( )

CHANGE HISTORY

First released in Issue 2.

NAME

wc — word, line, and byte or character count

SYNOPSIS

```
wc [−c|−m][−lw][file...]
```

DESCRIPTION

The *wc* utility shall read one or more input files and, by default, write the number of <newline>s, words, and bytes contained in each input file to the standard output.

The utility also shall write a total count for all named files, if more than one input file is specified.

The *wc* utility shall consider a *word* to be a non-zero-length string of characters delimited by white space.

OPTIONS

The *wc* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−c	Write to the standard output the number of bytes in each input file.
−l	Write to the standard output the number of <newline>s in each input file.
−m	Write to the standard output the number of characters in each input file.
−w	Write to the standard output the number of words in each input file.

When any option is specified, *wc* shall report only the information requested by the specified options.

OPERANDS

The following operand shall be supported:

file	A pathname of an input file. If no *file* operands are specified, the standard input shall be used.

STDIN

The standard input shall be used only if no *file* operands are specified. See the INPUT FILES section.

INPUT FILES

The input files may be of any type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *wc*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and which characters are defined as white space characters.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

XSI *NLSPATH* Determine the location of message catalogs for the processing of *LC_MESSAGES.*

ASYNCHRONOUS EVENTS

Default.

STDOUT

By default, the standard output shall contain an entry for each input file of the form:

```
"%d %d %d %s\n", <newlines>, <words>, <bytes>, <file>
```

If the **−m** option is specified, the number of characters shall replace the <*bytes*> field in this format.

If any options are specified and the **−l** option is not specified, the number of <newline>s shall not be written.

If any options are specified and the **−w** option is not specified, the number of words shall not be written.

If any options are specified and neither **−c** nor **−m** is specified, the number of bytes or characters shall not be written.

If no input *file* operands are specified, no name shall be written and no <blank>s preceding the pathname shall be written.

If more than one input *file* operand is specified, an additional line shall be written, of the same format as the other lines, except that the word **total** (in the POSIX locale) shall be written instead of a pathname and the total of each column shall be written as appropriate. Such an additional line, if any, is written at the end of the output.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The −**m** option is not a switch, but an option at the same level as −**c**. Thus, to produce the full default output with character counts instead of bytes, the command required is:

```
wc −mlw
```

EXAMPLES

None.

RATIONALE

The output file format pseudo-*printf*( ) string differs from the System V version of *wc*:

```
"%7d%7d%7d %s\n"
```

which produces possibly ambiguous and unparsable results for very large files, as it assumes no number shall exceed six digits.

Some historical implementations use only <space>, <tab>, and <newline> as word separators. The equivalent of the ISO C standard *isspace*( ) function is more appropriate.

The −**c** option stands for ''character'' count, even though it counts bytes. This stems from the sometimes erroneous historical view that bytes and characters are the same size. Due to international requirements, the −**m** option (reminiscent of ''multi-byte'') was added to obtain actual character counts.

Early proposals only specified the results when input files were text files. The current specification more closely matches historical practice. (Bytes, words, and <newline>s are counted separately and the results are written when an end-of-file is detected.)

Historical implementations of the *wc* utility only accepted one argument to specify the options −**c**, −**l**, and −**w**. Some of them also had multiple occurrences of an option cause the corresponding count to be written multiple times and had the order of specification of the options affect the order of the fields on output, but did not document either of these. Because common usage either specifies no options or only one option, and because none of this was documented, the changes required by this volume of IEEE Std 1003.1-200x should not break many historical applications (and do not break any historical conforming applications).

FUTURE DIRECTIONS

None.

SEE ALSO

cksum

CHANGE HISTORY

First released in Issue 2.

NAME

what — identify SCCS files (**DEVELOPMENT**)

SYNOPSIS

XSI
```
what [−s] file...
```

DESCRIPTION

The *what* utility shall search the given files for all occurrences of the pattern that *get* (see *get* (on page 2675)) substitutes for the %**Z**% keyword (`"@(#)"`) and shall write to standard output what follows until the first occurrence of one of the following:

```
"    >    newline    \    NUL
```

OPTIONS

The *what* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following option shall be supported:

−s	Quit after finding the first occurrence of the pattern in each file.

OPERANDS

The following operands shall be supported:

file	A pathname of a file to search.

STDIN

Not used.

INPUT FILES

The input files shall be of any file type.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *what*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output shall consist of the following for each *file* operand:

```
"%s:\n\t%s\n", <pathname>, <identification string>
```

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Any matches were found.
1	Otherwise.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *what* utility is intended to be used in conjunction with the SCCS command *get*, which automatically inserts identifying information, but it can also be used where the information is inserted by any other means.

When the string `"@(#)"` is included in a library routine in a shared library, it might not be found in an **a.out** file using that library routine.

EXAMPLES

If the C-language program in file **f.c** contains:

```
char ident[] = "@(#)identification information";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command:

```
what f.c f.o a.out
```

writes:

```
f.c:
    identification information
    ...
f.o:
    identification information
    ...
a.out:
    identification information
    ...
```

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

get

CHANGE HISTORY

First released in Issue 2.

NAME

who — display who is on the system

SYNOPSIS

```
UP   who [−mTu]

XSI  who [−mu]−s[−bHlprt][file]
     who [−mTu][−abdHlprt][file]
     who −q [file]
     who am i
     who am I
```

DESCRIPTION

The *who* utility shall list various pieces of information about accessible users. The domain of accessibility is implementation-defined.

XSI Based on the options given, *who* can also list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process ID of the command interpreter for each current system user.

OPTIONS

The *who* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported. The metavariables, such as <*line*>, refer to fields described in the STDOUT section.

XSI	**−a**	Process the implementation-defined database or named file with the **−b**, **−d**, **−l**, **−p**, **−r**, **−t**, **−T** and **−u** options turned on.
XSI	**−b**	Write the time and date of the last reboot.
XSI	**−d**	Write a list of all processes that have expired and not been respawned by the *init* system process. The <*exit*> field shall appear for dead processes and contain the termination and exit values of the dead process. This can be useful in determining why a process terminated.
XSI	**−H**	Write column headings above the regular output.
XSI	**−l**	(The letter ell.) List only those lines on which the system is waiting for someone to login. The <*name*> field shall be **LOGIN** in such cases. Other fields shall be the same as for user entries except that the <*state*> field does not exist.
	−m	Output only information about the current terminal.
XSI	**−p**	List any other process that is currently active and has been previously spawned by *init*.
XSI	**−q**	(Quick.) List only the names and the number of users currently logged on. When this option is used, all other options shall be ignored.
XSI	**−r**	Write the current *run-level* of the *init* process.
XSI	**−s**	List only the <*name*>, <*line*>, and <*time*> fields. This is the default case.
XSI	**−t**	Indicate the last change to the system clock.

−T	Show the state of each terminal, as described in the STDOUT section.
−u	Write ‘‘idle time’’ for each displayed user in addition to any other information. The idle time is the time since any activity occurred on the user’s terminal. The method of determining this is unspecified. XSI This option shall list only those users who are currently logged in. The <*name*> is the user’s login name. The <*line*> is the name of the line as found in the directory **/dev**. The <*time*> is the time that the user logged in. The <*activity*> is the number of hours and minutes since activity last occurred on that particular line. A dot indicates that the terminal has seen activity in the last minute and is therefore ‘‘current’’. If more than twenty-four hours have elapsed or the line has not been used since boot time, the entry shall be marked <*old*>. This field is useful when trying to determine whether a person is working at the terminal or not. The <*pid*> is the process ID of the user’s login process.

OPERANDS

XSI The following operands shall be supported:

am i, **am I**	In the POSIX locale, limit the output to describing the invoking user, equivalent to the −**m** option. The **am** and **i** or **I** must be separate arguments.
file	Specify a pathname of a file to substitute for the implementation-defined database of logged-on users that *who* uses by default.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *who*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
LC_TIME	Determine the locale used for the format and contents of the date and time strings.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*
TZ	Determine the timezone used when writing date and time information. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *who* utility shall write its default format to the standard output in an implementation-defined format, subject only to the requirement of containing the information described above.

XSI OF XSI-conformant systems shall write the default information to the standard output in the following general format:

`<name>[<state>]<line><time>[<activity>][<pid>][<comment>][<exit>]`

The following format shall be used for the **−T** option:

```
"%s %c %s %s\n" <name>, <terminal state>, <terminal name>,
    <time of login>
```

where <*terminal state*> is one of the following characters:

+	The terminal allows write access to other users.
–	The terminal denies write access to other users.
?	The terminal write-access state cannot be determined.

In the POSIX locale, the <*time of login*> shall be equivalent in format to the output of:

```
date +"%b %e %H:%M"
```

If the **−u** option is used with **−T**, the idle time shall be added to the end of the previous format in an unspecified format.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The name *init* used for the system process is the most commonly used on historical systems, but it may vary.

The ''domain of accessibility'' referred to is a broad concept that permits interpretation either on a very secure basis or even to allow a network-wide implementation like the historical *rwho*.

EXAMPLES

None.

RATIONALE

Due to differences between historical implementations, the base options provided were a compromise to allow users to work with those functions. The standard developers also considered removing all the options, but felt that these options offered users valuable functionality. Additional options to match historical systems are available on XSI-conformant systems.

It is recognized that the *who* command may be of limited usefulness, especially in a multi-level secure environment. The standard developers considered, however, that having some standard method of determining the ''accessibility'' of other users would aid user portability.

No format was specified for the default *who* output for systems not supporting the XSI Extension. In such a user-oriented command, designed only for human use, this was not considered to be a deficiency.

The format of the terminal name is unspecified, but the descriptions of *ps*, *talk*, and *write* require that they use the same format.

It is acceptable for an implementation to produce no output for an invocation of *who* **mil**.

FUTURE DIRECTIONS

None.

SEE ALSO

mesg

CHANGE HISTORY

First released in Issue 2.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The *TZ* entry is added to the ENVIRONMENT VARIABLES section.

NAME

write — write to another user

SYNOPSIS

UP
```
write user_name [terminal]
```

DESCRIPTION

The *write* utility shall read lines from the user's standard input and write them to the terminal of another user. When first invoked, it shall write the message:

```
Message from sender-login-id (sending-terminal) [date]...
```

to *user_name*. When it has successfully completed the connection, the sender's terminal shall be alerted twice to indicate that what the sender is typing is being written to the recipient's terminal.

If the recipient wants to reply, this can be accomplished by typing:

```
write sender-login-id [sending-terminal]
```

upon receipt of the initial message. Whenever a line of input as delimited by a NL, EOF, or EOL special character (see the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface) is accumulated while in canonical input mode, the accumulated data shall be written on the other user's terminal. Characters shall be processed as follows:

- Typing <alert> shall write the alert character to the recipient's terminal.
- Typing the erase and kill characters shall affect the sender's terminal in the manner described by the **termios** interface in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface.
- Typing the interrupt or end-of-file characters shall cause *write* to write an appropriate message (`"EOT\n"` in the POSIX locale) to the recipient's terminal and exit.
- Typing characters from *LC_CTYPE* classifications **print** or **space** shall cause those characters to be sent to the recipient's terminal.
- When and only when the *stty* **iexten** local mode is enabled, the existence and processing of additional special control characters and multi-byte or single-byte functions is implementation-defined.
- Typing other non-printable characters shall cause implementation-defined sequences of printable characters to be written to the recipient's terminal.

To write to a user who is logged in more than once, the *terminal* argument can be used to indicate which terminal to write to; otherwise, the recipient's terminal is selected in an implementation-defined manner and an informational message is written to the sender's standard output, indicating which terminal was chosen.

Permission to be a recipient of a *write* message can be denied or granted by use of the *mesg* utility. However, a user's privilege may further constrain the domain of accessibility of other users' terminals. The *write* utility shall fail when the user lacks the appropriate privileges to perform the requested action.

OPTIONS

None.

OPERANDS

The following operands shall be supported:

user_name	Login name of the person to whom the message shall be written. The application shall ensure that this operand is of the form returned by the *who* utility.
terminal	Terminal identification in the same format provided by the *who* utility.

STDIN

Lines to be copied to the recipient's terminal is read from standard input.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *write*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files). If the recipient's locale does not use an *LC_CTYPE* equivalent to the sender's, the results are undefined.
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If an interrupt signal is received, *write* shall write an appropriate message on the recipient's terminal and exits with a status of zero. It shall take the standard action for all other signals.

STDOUT

An informational message shall be written to standard output if a recipient is logged in more than once.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

The recipient's terminal is used for output.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	The addressed user is not logged on or the addressed user denies permission.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *talk* utility is considered by some users to be a more usable utility on full-screen terminals.

EXAMPLES

None.

RATIONALE

The *write* utility was included in this volume of IEEE Std 1003.1-200x since it can be implemented on all terminal types. The standard developers considered the *talk* utility, which cannot be implemented on certain terminals, to be a ‘‘better’’ communications interface. Both of these programs are in widespread use on historical implementations. Therefore, the standard developers decided that both utilities should be specified.

The format of the terminal name is unspecified, but the descriptions of *ps*, *talk*, *who*, and *write* require that they all use or accept the same format.

FUTURE DIRECTIONS

None.

SEE ALSO

mesg, *talk*, *who*, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the User Portability Utilities option.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

xargs — construct argument lists and invoke utility

SYNOPSIS

XSI
```
xargs [–t][–p]][–E eofstr][–I replstr][–L number][–n number [–x]]
    [–s size][utility [argument...]]
```

DESCRIPTION

The *xargs* utility shall construct a command line consisting of the *utility* and *argument* operands specified followed by as many arguments read in sequence from standard input as fit in length and number constraints specified by the options. The *xargs* utility shall then invoke the constructed command line and wait for its completion. This sequence shall be repeated until one of the following occurs:

- An end-of-file condition is detected on standard input.
- The logical end-of-file string (see the **–E** *eofstr* option) is found on standard input after double-quote processing, apostrophe processing, and backslash escape processing (see next paragraph).
- An invocation of a constructed command line returns an exit status of 255.

The application shall ensure that arguments in the standard input are separated by unquoted <blank>s, unescaped <blank>s or <newline>s. A string of zero or more non-double-quote (`'"'`) characters and non-<newline>s can be quoted by enclosing them in double-quotes. A string of zero or more non-apostrophe (`'\''`) characters and non-<newline>s can be quoted by enclosing them in apostrophes. Any unquoted character can be escaped by preceding it with a backslash. The utility named by *utility* shall be executed one or more times until the end-of-file is reached or the logical end-of file string is found. The results are unspecified if the utility named by *utility* attempts to read from its standard input.

The generated command line length shall be the sum of the size in bytes of the utility name and each argument treated as strings, including a null byte terminator for each of these strings. The *xargs* utility shall limit the command line length such that when the command line is invoked, the combined argument and environment lists (see the *exec* family of functions in the System Interfaces volume of IEEE Std 1003.1-200x) shall not exceed {ARG_MAX}–2 048 bytes. Within this constraint, if neither the **–n** nor the **–s** option is specified, the default command line length shall be at least {LINE_MAX}.

OPTIONS

The *xargs* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

–E *eofstr* Use *eofstr* as the logical end-of-file string. If **–E** is not specified, it is unspecified whether the logical end-of-file string is the underscore character (`'_'`) or the end-of-file string capability is disabled. When *eofstr* is the null string, the logical end-of-file string capability shall be disabled and underscore characters shall be taken literally.

XSI **–I** *replstr* Insert mode: *utility* is executed for each line from standard input, taking the entire line as a single argument, inserting it in *arguments* for each occurrence of *replstr*. A maximum of five arguments in *arguments* can each contain one or more instances of *replstr*. Any <blank>s at the beginning of each line shall be ignored. Constructed arguments cannot grow larger than 255 bytes. Option **–x** shall be forced on.

XSI **–L** *number* The *utility* shall be executed for each non-empty *number* lines of arguments from standard input. The last invocation of *utility* shall be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first <newline> unless the last character of the line is a <blank>; a trailing <blank> signals continuation to the next non-empty line, inclusive. The **–L** and **–n** options are mutually-exclusive; the last one specified shall take effect.

–n *number* Invoke *utility* using as many standard input arguments as possible, up to *number* (a positive decimal integer) arguments maximum. Fewer arguments shall be used if:

- The command line length accumulated exceeds the size specified by the **–s** option (or {LINE_MAX} if there is no **–s** option).
- The last iteration has fewer than but not zero, operands remaining.

–p Prompt mode: the user is asked whether to execute *utility* at each invocation. Trace mode (**–t**) is turned on to write the command instance to be executed, followed by a prompt to standard error. An affirmative response read from **/dev/tty** shall execute the command; otherwise, that particular invocation of *utility* shall be skipped.

–s *size* Invoke *utility* using as many standard input arguments as possible yielding a command line length less than *size* (a positive decimal integer) bytes. Fewer arguments shall be used if:

- The total number of arguments exceeds that specified by the **–n** option.
- XSI The total number of lines exceeds that specified by the **–L** option.
- End-of-file is encountered on standard input before *size* bytes are accumulated.

Values of *size* up to at least {LINE_MAX} bytes shall be supported, provided that the constraints specified in the DESCRIPTION are met. It shall not be considered an error if a value larger than that supported by the implementation or exceeding the constraints specified in the DESCRIPTION is given; *xargs* shall use the largest value it supports within the constraints.

–t Enable trace mode. Each generated command line shall be written to standard error just prior to invocation.

–x Terminate if a command line containing *number* arguments (see the **–n** option above) XSI or *number* lines (see the **–L** option above) will not fit in the implied or specified size (see the **–s** option above).

OPERANDS

The following operands shall be supported:

utility The name of the utility to be invoked, found by search path using the *PATH* environment variable, described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables. If *utility* is omitted, the default shall be the *echo* utility. If the *utility* operand names any of the special built-in utilities in Section 2.14 (on page 2266), the results are undefined.

argument An initial option or operand for the invocation of *utility*.

STDIN

The standard input shall be a text file. The results are unspecified if an end-of-file condition is detected immediately following an escaped <newline>.

INPUT FILES

The file **/dev/tty** shall be used to read responses required by the −**p** option.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *xargs*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_COLLATE	Determine the locale for the behavior of ranges, equivalence classes and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files) and the behavior of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.
LC_MESSAGES	Determine the locale for the processing of affirmative responses and that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES*.
PATH	Determine the location of *utility*, as described in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used for diagnostic messages and the −**t** and −**p** options. If the −**t** option is specified, the *utility* and its constructed argument list shall be written to standard error, as it will be invoked, prior to invocation. If −**p** is specified, a prompt of the following format shall be written (in the POSIX locale):

```
"?..."
```

at the end of the line of the output from −**t**.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0	All invocations of *utility* returned exit status zero.
1-125	A command line meeting the specified requirements could not be assembled, one or more of the invocations of *utility* returned a non-zero exit status, or some other error occurred.
126	The utility specified by *utility* was found but could not be invoked.
127	The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS

If a command line meeting the specified requirements cannot be assembled, the utility cannot be invoked, an invocation of the utility is terminated by a signal, or an invocation of the utility exits with exit status 255, the *xargs* utility shall write a diagnostic message and exit without processing any remaining input.

APPLICATION USAGE

The 255 exit status allows a utility being used by *xargs* to tell *xargs* to terminate if it knows no further invocations using the current data stream will succeed. Thus, *utility* should explicitly *exit* with an appropriate value to avoid accidentally returning with 255.

Note that input is parsed as lines; <blank>s separate arguments. If *xargs* is used to bundle output of commands like *find dir* −**print** or *ls* into commands to be executed, unexpected results are likely if any filenames contain any <blank>s or <newline>s. This can be fixed by using *find* to call a script that converts each file found into a quoted string that is then piped to *xargs*. Note that the quoting rules used by *xargs* are not the same as in the shell. They were not made consistent here because existing applications depend on the current rules and the shell syntax is not fully compatible with it. An easy rule that can be used to transform any string into a quoted form that *xargs* interprets correctly is to precede each character in the string with a backslash.

On implementations with a large value for {ARG_MAX}, *xargs* may produce command lines longer than {LINE_MAX}. For invocation of utilities, this is not a problem. If *xargs* is being used to create a text file, users should explicitly set the maximum command line length with the −**s** option.

The *command*, *env*, *nice*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

1. The following command combines the output of the parenthesised commands onto one line, which is then written to the end-of-file **log**:

   ```
   (logname; date; printf "%s\n" "$0 $*") | xargs >>log
   ```

2. The following command invokes *diff* with successive pairs of arguments originally typed as command line arguments (assuming there are no embedded <blank>s in the elements of the original argument list):

```
printf "%s\n" "$*" | xargs −n 2 −x diff
```

3. In the following commands, the user is asked which files in the current directory are to be archived. The files are archived into **arch**; *a*, one at a time, or *b*, many at a time.

```
a. ls | xargs −p −L 1 ar −r arch
```

```
b. ls | xargs −p −L 1 | xargs ar −r arch
```

4. The following executes with successive pairs of arguments originally typed as command line arguments:

```
echo $* | xargs −n 2 diff
```

5. On XSI-conformant systems, the following moves all files from directory **$1** to directory **$2**, and echoes each move command just before doing it:

```
ls $1 | xargs −I {} −t mv $1/{} $2/{}
```

RATIONALE

The *xargs* utility was usually found only in System V-based systems; BSD systems included an *apply* utility that provided functionality similar to *xargs* −**n** *number*. The SVID lists *xargs* as a software development extension. This volume of IEEE Std 1003.1-200x does not share the view that it is used only for development, and therefore it is not optional.

The classic application of the *xargs* utility is in conjunction with the *find* utility to reduce the number of processes launched by a simplistic use of the *find* −**exec** combination. The *xargs* utility is also used to enforce an upper limit on memory required to launch a process. With this basis in mind, this volume of IEEE Std 1003.1-200x selected only the minimal features required.

Although the 255 exit status is mostly an accident of historical implementations, it allows a utility being used by *xargs* to tell *xargs* to terminate if it knows no further invocations using the current data stream shall succeed. Any non-zero exit status from a utility falls into the 1-125 range when *xargs* exits. There is no statement of how the various non-zero utility exit status codes are accumulated by *xargs*. The value could be the addition of all codes, their highest value, the last one received, or a single value such as 1. Since no algorithm is arguably better than the others, and since many of the standard utilities say little more (portably) than ‘‘pass / fail’’, no new algorithm was invented.

Several other *xargs* options were withdrawn because simple alternatives already exist within this volume of IEEE Std 1003.1-200x. For example, the −**i** *replstr* option can be just as efficiently performed using a shell *for* loop. Since *xargs* calls an *exec* function with each input line, the −**i** option does not usually exploit the grouping capabilities of *xargs*.

The requirement that *xargs* never produce command lines such that invocation of *utility* is within 2 048 bytes of hitting the POSIX *exec* {ARG_MAX} limitations is intended to guarantee that the invoked utility has room to modify its environment variables and command line arguments and still be able to invoke another utility. Note that the minimum {ARG_MAX} allowed by the System Interfaces volume of IEEE Std 1003.1-200x is 4 096 bytes and the minimum value allowed by the this volume of IEEE Std 1003.1-200x is 2 048 bytes; therefore, the 2 048 bytes difference seems reasonable. Note, however, that *xargs* may never be able to invoke a utility if the environment passed in to *xargs* comes close to using {ARG_MAX} bytes.

The version of *xargs* required by this volume of IEEE Std 1003.1-200x is required to wait for the completion of the invoked command before invoking another command. This was done because historical scripts using *xargs* assumed sequential execution. Implementations wanting to provide parallel operation of the invoked utilities are encouraged to add an option enabling parallel invocation, but should still wait for termination of all of the children before *xargs* terminates normally.

The −**e** option was omitted from the ISO POSIX-2: 1993 standard in the belief that the *eofstr* option-argument was recognized only when it was on a line by itself and before quote and escape processing were performed, and that the logical end-of-file processing was only enabled if a −**e** option was specified. In that case, a simple *sed* script could be used to duplicate the −**e** functionality. Further investigation revealed that:

- The logical end-of-file string was checked for after quote and escape processing, making a *sed* script that provided equivalent functionality much more difficult to write.
- The default was to perform logical end-of-file processing with an underscore as the logical end-of-file string.

To correct this misunderstanding, the −**E** *eofstr* option was adopted from the X/Open Portability Guide. Users should note that the description of the −**E** option matches historical documentation of the −**e** option (which was not adopted because it did not support the Utility Syntax Guidelines), by saying that if *eofstr* is the null string, logical end-of-file processing is disabled. Historical implementations of *xargs* actually did not disable logical end-of-file processing; they treated a null argument found in the input as a logical end-of-file string. (A null *string* argument could be generated using single or double quotes (′ ′ or " "). Since this behavior was not documented historically, it is considered to be a bug.

FUTURE DIRECTIONS

None.

SEE ALSO

echo

CHANGE HISTORY

First released in Issue 2.

Issue 5

Second FUTURE DIRECTION added.

Issue 6

The obsolescent −**e**, −**i**, and −**l** options are removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The −**p** option is added.
- In the INPUT FILES section, the file **/dev/tty** is used to read responses required by the −**p** option.
- The STDERR section is updated to describe the −**p** option.

The description of the −**E** option is aligned with the ISO POSIX-2: 1993 standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

NAME

yacc — yet another compiler compiler (**DEVELOPMENT**)

SYNOPSIS

CD
```
yacc [−dltv][−b file_prefix][−p sym_prefix] grammar
```

DESCRIPTION

The *yacc* utility shall read a description of a context-free grammar in *file* and write C source code, conforming to the ISO C standard, to a code file, and optionally header information into a header file, in the current directory. The C code shall define a function and related routines and macros for an automaton that executes a parsing algorithm meeting the requirements in **Algorithms** (on page 3272).

The form and meaning of the grammar are described in the EXTENDED DESCRIPTION section.

The C source code and header file shall be produced in a form suitable as input for the C compiler (see *c99* (on page 2413)).

OPTIONS

The *yacc* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

−**b** *file_prefix*	Use *file_prefix* instead of **y** as the prefix for all output filenames. The code file **y.tab.c**, the header file **y.tab.h** (created when −**d** is specified), and the description file **y.output** (created when −**v** is specified), shall be changed to *file_prefix***.tab.c**, *file_prefix***.tab.h**, and *file_prefix***.output**, respectively.
−**d**	Write the header file; by default only the code file is written. The **#define** statements that associate the token codes assigned by *yacc* with the user-declared token names. This allows source files other than **y.tab.c** to access the token codes.
−**l**	Produce a code file that does not contain any **#line** constructs. If this option is not present, it is unspecified whether the code file or header file contains **#line** directives. This should only be used after the grammar and the associated actions are fully debugged.
−**p** *sym_prefix*	Use *sym_prefix* instead of **yy** as the prefix for all external names produced by *yacc*. The names affected shall include the functions *yyparse*( ), *yylex*( ), and *yyerror*( ), and the variables *yylval*, *yychar*, and *yydebug*. (In the remainder of this section, the six symbols cited are referenced using their default names only as a notational convenience.) Local names may also be affected by the −**p** option; however, the −**p** option shall not affect **#define** symbols generated by *yacc*.
−**t**	Modify conditional compilation directives to permit compilation of debugging code in the code file. Runtime debugging statements shall always be contained in the code file, but by default conditional compilation directives prevent their compilation.
−**v**	Write a file containing a description of the parser and a report of conflicts generated by ambiguities in the grammar.

OPERANDS

The following operand is required:

grammar	A pathname of a file containing instructions, hereafter called *grammar*, for which a parser is to be created. The format for the grammar is described in the EXTENDED

DESCRIPTION section.

STDIN

Not used.

INPUT FILES

The file *grammar* shall be a text file formatted as specified in the EXTENDED DESCRIPTION section.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *yacc*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
XSI *NLSPATH*	Determine the location of message catalogs for the processing of *LC_MESSAGES.*

The *LANG* and *LC_** variables affect the execution of the *yacc* utility as stated. The *main*() function defined in **Yacc Library** (on page 3272) shall call:

```
setlocale(LC_ALL, "")
```

and thus, the program generated by *yacc* also shall be affected by the contents of these variables at runtime.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

If shift/reduce or reduce/reduce conflicts are detected in *grammar*, *yacc* shall write a report of those conflicts to the standard error in an unspecified format.

Standard error shall also be used for diagnostic messages.

OUTPUT FILES

The code file, the header file, and the description file shall be text files. All are described in the following sections.

Code File

This file shall contain the C source code for the *yyparse*( ) function. It shall contain code for the various semantic actions with macro substitution performed on them as described in the EXTENDED DESCRIPTION section. It also shall contain a copy of the **#define** statements in the header file. If a **%union** declaration is used, the declaration for YYSTYPE shall be also included in this file.

Header File

The header file shall contain **#define** statements that associate the token numbers with the token names. This allows source files other than the code file to access the token codes. If a **%union** declaration is used, the declaration for YYSTYPE and an *extern YYSTYPE yylval* declaration shall be also included in this file.

Description File

The description file shall be a text file containing a description of the state machine corresponding to the parser, using an unspecified format. Limits for internal tables (see **Limits** (on page 3273)) shall also be reported, in an implementation-defined manner. (Some implementations may use dynamic allocation techniques and have no specific limit values to report.)

EXTENDED DESCRIPTION

The *yacc* command accepts a language that is used to define a grammar for a target language to be parsed by the tables and code generated by *yacc*. The language accepted by *yacc* as a grammar for the target language is described below using the *yacc* input language itself.

The input *grammar* includes rules describing the input structure of the target language and code to be invoked when these rules are recognized to provide the associated semantic action. The code to be executed shall appear as bodies of text that are intended to be C-language code. The C-language inclusions are presumed to form a correct function when processed by *yacc* into its output files. The code included in this way shall be executed during the recognition of the target language.

Given a grammar, the *yacc* utility generates the files described in the OUTPUT FILES section. The code file can be compiled and linked using *c99*. If the declaration and programs sections of the grammar file did not include definitions of *main*( ), *yylex*( ), and *yyerror*( ), the compiled output requires linking with externally supplied version of those functions. Default versions of *main*( ) and *yyerror*( ) are supplied in the *yacc* library and can be linked in by using the **–l y** operand to *c99*. The *yacc* library interfaces need not support interfaces with other than the default **yy** symbol prefix. The application provides the lexical analyzer function, *yylex*( ); the *lex* utility is specifically designed to generate such a routine.

Input Language

The application shall ensure that every specification file consists of three sections in order: *declarations*, *grammar rules*, and *programs*, separated by double percent signs ("%%"). The declarations and programs sections can be empty. If the latter is empty, the preceding "%%" mark separating it from the rules section can be omitted.

The input is free form text following the structure of the grammar defined below.

Lexical Structure of the Grammar

The <blank>s, <newline>s, and <form-feed>s shall be ignored, except that the application shall ensure that they do not appear in names or multi-character reserved symbols. Comments shall be enclosed in `"/* ... */"`, and can appear wherever a name is valid.

Names are of arbitrary length, made up of letters, periods (`'.'`), underscores (`'_'`), and non-initial digits. Uppercase and lowercase letters are distinct. Conforming applications shall not use names beginning in **yy** or **YY** since the *yacc* parser uses such names. Many of the names appear in the final output of *yacc*, and thus they should be chosen to conform with any additional rules created by the C compiler to be used. In particular they appear in **#define** statements.

A literal shall consist of a single character enclosed in single-quotes (`'\''`). All of the escape sequences supported for character constants by the ISO C standard shall be supported by *yacc*.

The relationship with the lexical analyzer is discussed in detail below.

The application shall ensure that the NUL character is not used in grammar rules or literals.

Declarations Section

The declarations section is used to define the symbols used to define the target language and their relationship with each other. In particular, much of the additional information required to resolve ambiguities in the context-free grammar for the target language is provided here.

Usually *yacc* assigns the relationship between the symbolic names it generates and their underlying numeric value. The declarations section makes it possible to control the assignment of these values.

It is also possible to keep semantic information associated with the tokens currently on the parse stack in a user-defined C-language **union**, if the members of the union are associated with the various names in the grammar. The declarations section provides for this as well.

The first group of declarators below all take a list of names as arguments. That list can optionally be preceded by the name of a C union member (called a *tag* below) appearing within `'<'` and `'>'`. (As an exception to the typographical conventions of the rest of this volume of IEEE Std 1003.1-200x, in this case <*tag*> does not represent a metavariable, but the literal angle bracket characters surrounding a symbol.) The use of *tag* specifies that the tokens named on this line shall be of the same C type as the union member referenced by *tag*. This is discussed in more detail below.

For lists used to define tokens, the first appearance of a given token can be followed by a positive integer (as a string of decimal digits). If this is done, the underlying value assigned to it for lexical purposes shall be taken to be that number.

The following declares *name* to be a token:

```
token [<tag>] name [number][name [number]]...
```

If *tag* is present, the C type for all tokens on this line shall be declared to be the type referenced by *tag*. If a positive integer, *number*, follows a *name*, that value shall be assigned to the token.

The following declares *name* to be a token, and assigns precedence to it:

```
%left [<tag>] name [number][name [number]]...
%right [<tag>] name [number][name [number]]...
```

One or more lines, each beginning with one of these symbols, can appear in this section. All tokens on the same line have the same precedence level and associativity; the lines are in order of increasing precedence or binding strength. **%left** denotes that the operators on that line are left associative, and **%right** similarly denotes right associative operators. If *tag* is present, it shall declare a C type for *name*s as described for **%token**.

The following declares *name* to be a token, and indicates that this cannot be used associatively:

```
%nonassoc [<tag>] name [number][name [number]]...
```

If the parser encounters associative use of this token it reports an error. If *tag* is present, it shall declare a C type for *name*s as described for **%token**.

The following declares that union member *name*s are non-terminals, and thus it is required to have a *tag* field at its beginning:

```
%type <tag> name...
```

Because it deals with non-terminals only, assigning a token number or using a literal is also prohibited. If this construct is present, *yacc* shall perform type checking; if this construct is not present, the parse stack shall hold only the **int** type.

Every name used in *grammar* not defined by a **%token**, **%left**, **%right**, or **%nonassoc** declaration is assumed to represent a non-terminal symbol. The *yacc* utility shall report an error for any non-terminal symbol that does not appear on the left side of at least one grammar rule.

Once the type, precedence, or token number of a name is specified, it shall not be changed. If the first declaration of a token does not assign a token number, *yacc* shall assign a token number. Once this assignment is made, the token number shall not be changed by explicit assignment.

The following declarators do not follow the previous pattern.

The following declares the non-terminal *name* to be the *start symbol*, which represents the largest, most general structure described by the grammar rules:

```
%start name
```

By default, it is the left-hand side of the first grammar rule; this default can be overridden with this declaration.

The following declares the *yacc* value stack to be a union of the various types of values desired:

```
%union { body of union (in C) }
```

By default, the values returned by actions (see below) and the lexical analyzer shall be of type **int**. The *yacc* utility keeps track of types, and it shall insert corresponding union member names in order to perform strict type checking of the resulting parser.

Alternatively, given that at least one <*tag*> construct is used, the union can be declared in a header file (which shall be included in the declarations section by using an **#include** construct within **%{** and **%}**), and a **typedef** used to define the symbol YYSTYPE to represent this union. The effect of **%union** is to provide the declaration of YYSTYPE directly from the *yacc* input.

C-language declarations and definitions can appear in the declarations section, enclosed by the following marks:

```
%{ ... %}
```

These statements shall be copied into the code file, and have global scope within it so that they can be used in the rules and program sections.

The application shall ensure that the declarations section is terminated by the token **%%**.

Grammar Rules in yacc

The rules section defines the context-free grammar to be accepted by the function *yacc* generates, and associates with those rules C-language actions and additional precedence information. The grammar is described below, and a formal definition follows.

The rules section is comprised of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

The symbol **A** represents a non-terminal name, and **BODY** represents a sequence of zero or more *name*s, *literal*s, and *semantic action*s that can then be followed by optional *precedence rule*s. Only the names and literals participate in the formation of the grammar; the semantic actions and precedence rules are used in other ways. The colon and the semicolon are *yacc* punctuation. If there are several successive grammar rules with the same left-hand side, the vertical bar `'|'` can be used to avoid rewriting the left-hand side; in this case the semicolon appears only after the last rule. The BODY part can be empty (or empty of names and literals) to indicate that the non-terminal symbol matches the empty string.

The *yacc* utility assigns a unique number to each rule. Rules using the vertical bar notation are distinct rules. The number assigned to the rule appears in the description file.

The elements comprising a BODY are:

name, *literal* These form the rules of the grammar: *name* is either a *token* or a *non-terminal*; *literal* stands for itself (less the lexically required quotation marks).

semantic action

With each grammar rule, the user can associate actions to be performed each time the rule is recognized in the input process. (Note that the word ‘‘action’’ can also refer to the actions of the parser—shift, reduce, and so on.)

These actions can return values and can obtain the values returned by previous actions. These values are kept in objects of type YYSTYPE (see **%union**). The result value of the action shall be kept on the parse stack with the left-hand side of the rule, to be accessed by other reductions as part of their right-hand side. By using the <*tag*> information provided in the declarations section, the code generated by *yacc* can be strictly type checked and contain arbitrary information. In addition, the lexical analyzer can provide the same kinds of values for tokens, if desired.

An action is an arbitrary C statement and as such can do input or output, call subprograms and alter external variables. An action is one or more C statements enclosed in curly braces `'{'` and `'}'`.

Certain pseudo-variables can be used in the action. These are macros for access to data structures known internally to *yacc*.

$$ The value of the action can be set by assigning it to $$. If type checking is enabled and the type of the value to be assigned cannot be determined, a diagnostic message may be generated.

$*number* This refers to the value returned by the component specified by the token *number* in the right side of a rule, reading from left to right; *number* can be zero or negative. If it is, it refers to the data associated with the name on the parser's stack preceding the leftmost symbol of the current rule. (That is, `"$0"` refers to the name immediately preceding the leftmost name in the current rule, to be found on the parser's stack and `"$−1"` refers to the symbol to *its* left.) If *number* refers to an element past the current point in the rule, or beyond the bottom of the stack, the result is undefined. If type checking is enabled and the type of the value to be assigned cannot be determined, a diagnostic message may be generated.

$<*tag*>*number*
These correspond exactly to the corresponding symbols without the *tag* inclusion, but allow for strict type checking (and preclude unwanted type conversions). The effect is that the macro is expanded to use *tag* to select an element from the YYSTYPE union (using *dataname.tag*). This is particularly useful if *number* is not positive.

$<*tag*>$ This imposes on the reference the type of the union member referenced by *tag*. This construction is applicable when a reference to a left context value occurs in the grammar, and provides *yacc* with a means for selecting a type.

Actions can occur anywhere in a rule (not just at the end); an action can access values returned by actions to its left, and in turn the value it returns can be accessed by actions to its right. An action appearing in the middle of a rule shall be equivalent to replacing the action with a new non-terminal symbol and adding an empty rule with that non-terminal symbol on the left-hand side. The semantic action associated with the new rule shall be equivalent to the original action. The use of actions within rules might introduce conflicts that would not otherwise exist.

By default, the value of a rule shall be the value of the first element in it. If the first element does not have a type (particularly in the case of a literal) and type checking is turned on by **%type** an error message shall result.

precedence The keyword **%prec** can be used to change the precedence level associated with a particular grammar rule. Examples of this are in cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences, or where the handling of an ambiguous if-else construction is necessary. The reserved symbol **%prec** can appear immediately after the body of the grammar rule and can be followed by a token name or a literal. It shall cause the precedence of the grammar rule to become that of the following token name or literal. The action for the rule as a whole can follow **%prec**.

If a program section follows, the application shall ensure that the grammar rules are terminated by **%%**.

Programs Section

The *programs* section can include the definition of the lexical analyzer *yylex*( ), and any other functions, for example those used in the actions specified in the grammar rules. It is unspecified whether the programs section precedes or follows the semantic actions in the output file; therefore, if the application contains any macro definitions and declarations intended to apply to the code in the semantic actions, it shall place them within `"%{ ... %}"` in the declarations section.

Input Grammar

The following input to *yacc* yields a parser for the input to *yacc*. This formal syntax takes precedence over the preceding text syntax description.

The lexical structure is defined less precisely; **Lexical Structure of the Grammar** (on page 3264) defines most terms. The correspondence between the previous terms and the tokens below is as follows.

IDENTIFIER	This corresponds to the concept of *name*, given previously. It also includes literals as defined previously.
C_IDENTIFIER	This is a name, and additionally it is known to be followed by a colon. A literal cannot yield this token.
NUMBER	A string of digits (a non-negative decimal integer).
TYPE, **LEFT**, **MARK**, **LCURL**, **RCURL**	These correspond directly to **%type**, **%left**, **%%**, **%{**, and **%}**.
{ . . . }	This indicates C-language source code, with the possible inclusion of '$' macros as discussed previously.

```
/* Grammar for the input to yacc. */
/* Basic entries. */
/* The following are recognized by the lexical analyzer. */

%token    IDENTIFIER      /* Includes identifiers and literals */
%token    C_IDENTIFIER    /* identifier (but not literal)
                             followed by a :. */
%token    NUMBER          /* [0-9][0-9]* */

/* Reserved words : %type=>TYPE %left=>LEFT, and so on */

%token    LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token    MARK            /* The %% mark. */
%token    LCURL           /* The %{ mark. */
%token    RCURL           /* The %} mark. */

/* 8-bit character literals stand for themselves; */
/* tokens have to be defined for multi-byte characters. */

%start    spec

%%

spec   : defs MARK rules tail
       ;
tail   : MARK
       {
         /* In this action, set up the rest of the file. */
       }
       | /* Empty; the second MARK is optional. */
       ;
defs   : /* Empty. */
       |    defs def
       ;
def    : START IDENTIFIER
       |    UNION
```

```
             {
               /* Copy union definition to output. */
             }
             |    LCURL
             {
               /* Copy C code to output file. */
             }
               RCURL
             |    rword tag nlist
             ;
      rword : TOKEN
             | LEFT
             | RIGHT
             | NONASSOC
             | TYPE
             ;
      tag   : /* Empty: union tag ID optional. */
             | '<' IDENTIFIER '>'
             ;
      nlist : nmno
             | nlist nmno
             ;
      nmno  : IDENTIFIER         /* Note: literal invalid with % type. */
             | IDENTIFIER NUMBER  /* Note: invalid with % type. */
             ;
      /* Rule section */
      rules : C_IDENTIFIER rbody prec
             | rules  rule
             ;
      rule  : C_IDENTIFIER rbody prec
             | '|' rbody prec
             ;
      rbody : /* empty */
             | rbody IDENTIFIER
             | rbody act
             ;
      act   : '{'
               {
                 /* Copy action, translate $$, and so on. */
               }
               '}'
             ;
      prec  : /* Empty */
             | PREC IDENTIFIER
             | PREC IDENTIFIER act
             | prec ';'
             ;
```

Conflicts

The parser produced for an input grammar may contain states in which conflicts occur. The conflicts occur because the grammar is not LALR(1). An ambiguous grammar always contains at least one LALR(1) conflict. The *yacc* utility shall resolve all conflicts, using either default rules or user-specified precedence rules.

Conflicts are either shift/reduce conflicts or reduce/reduce conflicts. A shift/reduce conflict is where, for a given state and lookahead symbol, both a shift action and a reduce action are possible. A reduce/reduce conflict is where, for a given state and lookahead symbol, reductions by two different rules are possible.

The rules below describe how to specify what actions to take when a conflict occurs. Not all shift/reduce conflicts can be successfully resolved this way because the conflict may be due to something other than ambiguity, so incautious use of these facilities can cause the language accepted by the parser to be much different from that which was intended. The description file shall contain sufficient information to understand the cause of the conflict. Where ambiguity is the reason either the default or explicit rules should be adequate to produce a working parser.

The declared precedences and associativities (see **Declarations Section** (on page 3264)) are used to resolve parsing conflicts as follows:

1. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** keyword is used, it overrides this default. Some grammar rules might not have both precedence and associativity.
2. If there is a shift/reduce conflict, and both the grammar rule and the input symbol have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and non-associative implies an error in the string being parsed.
3. When there is a shift/reduce conflict that cannot be resolved by rule 2, the shift is done. Conflicts resolved this way are counted in the diagnostic output described in **Error Handling**.
4. When there is a reduce/reduce conflict, a reduction is done by the grammar rule that occurs earlier in the input sequence. Conflicts resolved this way are counted in the diagnostic output described in **Error Handling**.

Conflicts resolved by precedence or associativity shall not be counted in the shift/reduce and reduce/reduce conflicts reported by *yacc* on either standard error or in the description file.

Error Handling

The token **error** shall be reserved for error handling. The name **error** can be used in grammar rules. It indicates places where the parser can recover from a syntax error. The default value of **error** shall be 256. Its value can be changed using a **%token** declaration. The lexical analyzer should not return the value of **error**.

The parser shall detect a syntax error when it is in a state where the action associated with the lookahead symbol is **error**. A semantic action can cause the parser to initiate error handling by executing the macro YYERROR. When YYERROR is executed, the semantic action passes control back to the parser. YYERROR cannot be used outside of semantic actions.

When the parser detects a syntax error, it normally calls *yyerror*( ) with the character string `"syntax error"` as its argument. The call shall not be made if the parser is still recovering from a previous error when the error is detected. The parser is considered to be recovering from a previous error until the parser has shifted over at least three normal input symbols since the last error was detected or a semantic action has executed the macro *yyerrok*. The parser shall not call *yyerror*( ) when YYERROR is executed.

The macro function YYRECOVERING shall return 1 if a syntax error has been detected and the parser has not yet fully recovered from it. Otherwise, zero shall be returned.

When a syntax error is detected by the parser, the parser shall check if a previous syntax error has been detected. If a previous error was detected, and if no normal input symbols have been shifted since the preceding error was detected, the parser checks if the lookahead symbol is an endmarker (see **Interface to the Lexical Analyzer**). If it is, the parser shall return with a non-zero value. Otherwise, the lookahead symbol shall be discarded and normal parsing shall resume.

When YYERROR is executed or when the parser detects a syntax error and no previous error has been detected, or at least one normal input symbol has been shifted since the previous error was detected, the parser shall pop back one state at a time until the parse stack is empty or the current state allows a shift over **error**. If the parser empties the parse stack, it shall return with a non-zero value. Otherwise, it shall shift over **error** and then resume normal parsing. If the parser reads a lookahead symbol before the error was detected, that symbol shall still be the lookahead symbol when parsing is resumed.

The macro *yyerrok* in a semantic action shall cause the parser to act as if it has fully recovered from any previous errors. The macro *yyclearin* shall cause the parser to discard the current lookahead token. If the current lookahead token has not yet been read, *yyclearin* shall have no effect.

The macro YYACCEPT shall cause the parser to return with the value zero. The macro YYABORT shall cause the parser to return with a non-zero value.

Interface to the Lexical Analyzer

The *yylex*( ) function is an integer-valued function that returns a *token number* representing the kind of token read. If there is a value associated with the token returned by *yylex*( ) (see the discussion of *tag* above), it shall be assigned to the external variable *yylval*.

If the parser and *yylex*( ) do not agree on these token numbers, reliable communication between them cannot occur. For (single-byte character) literals, the token is simply the numeric value of the character in the current character set. The numbers for other tokens can either be chosen by *yacc*, or chosen by the user. In either case, the **#define** construct of C is used to allow *yylex*( ) to return these numbers symbolically. The **#define** statements are put into the code file, and the header file if that file is requested. The set of characters permitted by *yacc* in an identifier is larger than that permitted by C. Token names found to contain such characters shall not be included in the **#define** declarations.

If the token numbers are chosen by *yacc*, the tokens other than literals shall be assigned numbers greater than 256, although no order is implied. A token can be explicitly assigned a number by following its first appearance in the declarations section with a number. Names and literals not defined this way retain their default definition. All token numbers assigned by *yacc* shall be unique and distinct from the token numbers used for literals and user-assigned tokens. If duplicate token numbers cause conflicts in parser generation, *yacc* shall report an error; otherwise, it is unspecified whether the token assignment is accepted or an error is reported.

The end of the input is marked by a special token called the *endmarker*, which has a token number that is zero or negative. (These values are invalid for any other token.) All lexical analyzers shall return zero or negative as a token number upon reaching the end of their input. If the tokens up to, but excluding, the endmarker form a structure that matches the start symbol, the parser shall accept the input. If the endmarker is seen in any other context, it shall be considered an error.

Completing the Program

In addition to *yyparse*( ) and *yylex*( ), the functions *yyerror*( ) and *main*( ) are required to make a complete program. The application can supply *main*( ) and *yyerror*( ), or those routines can be obtained from the *yacc* library.

Yacc Library

The following functions shall appear only in the *yacc* library accessible through the **–l y** operand to *c99*; they can therefore be redefined by a conforming application: |

int *main*(**void**)
This function shall call *yyparse*( ) and exit with an unspecified value. Other actions within this function are unspecified.

int *yyerror*(**const char** **s*)
This function shall write the NUL-terminated argument to standard error, followed by a <newline>.

The order of the **–l y** and **–l l** operands given to *c99* is significant; the application shall either provide its own *main*( ) function or ensure that **–l y** precedes **–l l**.

Debugging the Parser

The parser generated by *yacc* shall have diagnostic facilities in it that can be optionally enabled at either compile time or at runtime (if enabled at compile time). The compilation of the runtime debugging code is under the control of YYDEBUG, a preprocessor symbol. If YYDEBUG has a non-zero value, the debugging code shall be included. If its value is zero, the code shall not be included.

In parsers where the debugging code has been included, the external **int** *yydebug* can be used to turn debugging on (with a non-zero value) and off (zero value) at runtime. The initial value of *yydebug* shall be zero.

When **–t** is specified, the code file shall be built such that, if YYDEBUG is not already defined at compilation time (using the *c99* **–D** YYDEBUG option, for example), YYDEBUG shall be set explicitly to 1. When **–t** is not specified, the code file shall be built such that, if YYDEBUG is not already defined, it shall be set explicitly to zero.

The format of the debugging output is unspecified but includes at least enough information to determine the shift and reduce actions, and the input symbols. It also provides information about error recovery.

Algorithms

The parser constructed by *yacc* implements an LALR(1) parsing algorithm as documented in the literature. It is unspecified whether the parser is table-driven or direct-coded.

A parser generated by *yacc* shall never request an input symbol from *yylex*( ) while in a state where the only actions other than the error action are reductions by a single rule.

The literature of parsing theory defines these concepts.

Limits

The *yacc* utility may have several internal tables. The minimum maximums for these tables are shown in the following table. The exact meaning of these values is implementation-defined. The implementation shall define the relationship between these values and between them and any error messages that the implementation may generate should it run out of space for any internal structure. An implementation may combine groups of these resources into a single pool as long as the total available to the user does not fall below the sum of the sizes specified by this section.

Table 4-22 Internal Limits in *yacc*

Limit	Minimum Maximum	Description
{NTERMS}	126	Number of tokens.
{NNONTERM}	200	Number of non-terminals.
{NPROD}	300	Number of rules.
{NSTATES}	600	Number of states.
{MEMSIZE}	5 200	Length of rules. The total length, in names (tokens and non-terminals), of all the rules of the grammar. The left-hand side is counted for each rule, even if it is not explicitly repeated, as specified in **Grammar Rules in yacc** (on page 3266).
{ACTSIZE}	4 000	Number of actions. ''Actions'' here (and in the description file) refer to parser actions (shift, reduce, and so on) not to semantic actions defined in **Grammar Rules in yacc** (on page 3266).

EXIT STATUS

The following exit values shall be returned:

0	Successful completion.
>0	An error occurred.

CONSEQUENCES OF ERRORS

If any errors are encountered, the run is aborted and *yacc* exits with a non-zero status. Partial code files and header files files may be produced. The summary information in the description file always shall be produced if the **–v** flag is present.

APPLICATION USAGE

Historical implementations experience name conflicts on the names **yacc.tmp**, **yacc.acts**, **yacc.debug**, **y.tab.c**, **y.tab.h**, and **y.output** if more than one copy of *yacc* is running in a single directory at one time. The **–b** option was added to overcome this problem. The related problem of allowing multiple *yacc* parsers to be placed in the same file was addressed by adding a **–p** option to override the previously hard-coded **yy** variable prefix.

The description of the **–p** option specifies the minimal set of function and variable names that cause conflict when multiple parsers are linked together. YYSTYPE does not need to be changed. Instead, the programmer can use **–b** to give the header files for different parsers different names, and then the file with the *yylex*( ) for a given parser can include the header for that parser. Names such as *yyclearerr* do not need to be changed because they are used only in the actions; they do not have linkage. It is possible that an implementation has other names, either internal ones for implementing things such as *yyclearerr*, or providing non-standard features that it wants to change with **–p**.

Unary operators that are the same token as a binary operator in general need their precedence adjusted. This is handled by the **%prec** advisory symbol associated with the particular grammar rule defining that unary operator. (See **Grammar Rules in yacc** (on page 3266).) Applications are not required to use this operator for unary operators, but the grammars that do not require it are rare.

EXAMPLES

Access to the *yacc* library is obtained with library search operands to *c99*. To use the *yacc* library *main*( ):

```
c99 y.tab.c −l y
```

Both the *lex* library and the *yacc* library contain *main*( ). To access the *yacc main*( ):

```
c99 y.tab.c lex.yy.c −l y −l l
```

This ensures that the *yacc* library is searched first, so that its *main*( ) is used.

The historical *yacc* libraries have contained two simple functions that are normally coded by the application programmer. These functions are similar to the following code:

```
#include <locale.h>
int main(void)
{
    extern int yyparse();

    setlocale(LC_ALL, "");

    /* If the following parser is one created by lex, the
       application must be careful to ensure that LC_CTYPE
       and LC_COLLATE are set to the POSIX locale. */
    (void) yyparse();
    return (0);
}

#include <stdio.h>

int yyerror(const char *msg)
{
    (void) fprintf(stderr, "%s\n", msg);
    return (0);
}
```

RATIONALE

The references in **Referenced Documents** (on page xx) may be helpful in constructing the parser generator. The referenced DeRemer and Pennello article (along with the works it references) describes a technique to generate parsers that conform to this volume of IEEE Std 1003.1-200x. Work in this area continues to be done, so implementors should consult current literature before doing any new implementations. The original Knuth article is the theoretical basis for this kind of parser, but the tables it generates are impractically large for reasonable grammars and should not be used. The ‘‘equivalent to’’ wording is intentional to assure that the best tables that are LALR(1) can be generated.

There has been confusion between the class of grammars, the algorithms needed to generate parsers, and the algorithms needed to parse the languages. They are all reasonably orthogonal. In particular, a parser generator that accepts the full range of LR(1) grammars need not generate a table any more complex than one that accepts SLR(1) (a relatively weak class of LR grammars) for a grammar that happens to be SLR(1). Such an implementation need not recognize the case, either; table compression can yield the SLR(1) table (or one even smaller than that) without recognizing that the grammar is SLR(1). The speed of an LR(1) parser for any class is dependent more upon the table representation and compression (or the code generation if a direct parser is generated) than upon the class of grammar that the table generator handles.

The speed of the parser generator is somewhat dependent upon the class of grammar it handles. However, the original Knuth article algorithms for constructing LR parsers was judged by its author to be impractically slow at that time. Although full LR is more complex than LALR(1), as computer speeds and algorithms improve, the difference (in terms of acceptable wall-clock execution time) is becoming less significant.

Potential authors are cautioned that the referenced DeRemer and Pennello article previously cited identifies a bug (an over-simplification of the computation of LALR(1) lookahead sets) in some of the LALR(1) algorithm statements that preceded it to publication. They should take the time to seek out that paper, as well as current relevant work, particularly Aho's.

The −**b** option was added to provide a portable method for permitting *yacc* to work on multiple separate parsers in the same directory. If a directory contains more than one *yacc* grammar, and both grammars are constructed at the same time (by, for example, a parallel *make* program), conflict results. While the solution is not historical practice, it corrects a known deficiency in historical implementations. Corresponding changes were made to all sections that referenced the filenames **y.tab.c** (now ‘‘the code file’’), **y.tab.h** (now ‘‘the header file’’), and **y.output** (now ‘‘the description file’’).

The grammar for *yacc* input is based on System V documentation. The textual description shows there that the `';'` is required at the end of the rule. The grammar and the implementation do not require this. (The use of **C_IDENTIFIER** causes a reduce to occur in the right place.)

Also, in that implementation, the constructs such as **%token** can be terminated by a semicolon, but this is not permitted by the grammar. The keywords such as **%token** can also appear in uppercase, which is again not discussed. In most places where `'%'` is used, `'\'` can be substituted, and there are alternate spellings for some of the symbols (for example, **%LEFT** can be "%<" or even "\<").

Historically, <*tag*> can contain any characters except `'>'`, including white space, in the implementation. However, since the *tag* must reference a ISO C standard union member, in practice conforming implementations need to support only the set of characters for ISO C standard identifiers in this context.

Some historical implementations are known to accept actions that are terminated by a period. Historical implementations often allow `'$'` in names. A conforming implementation does not need to support either of these behaviors.

Deciding when to use **%prec** illustrates the difficulty in specifying the behavior of *yacc*. There may be situations in which the *grammar* is not, strictly speaking, in error, and yet *yacc* cannot interpret it unambiguously. The resolution of ambiguities in the grammar can in many instances be resolved by providing additional information, such as using **%type** or **%union** declarations. It is often easier and it usually yields a smaller parser to take this alternative when it is appropriate.

The size and execution time of a program produced without the runtime debugging code is usually smaller and slightly faster in historical implementations.

Statistics messages from several historical implementations include the following types of information:

```
n/512 terminals, n/300 non-terminals
n/600 grammar rules, n/1500 states
n shift/reduce, n reduce/reduce conflicts reported
```

```
n/350 working sets used
Memory: states, etc. n/15 000, parser n/15 000
n/600 distinct lookahead sets
n extra closures
n shift entries, n exceptions
n goto entries
n entries saved by goto default
Optimizer space used: input n/15 000, output n/15 000
n table entries, n zero
Maximum spread: n, Maximum offset: n
```

The report of internal tables in the description file is left implementation-defined because all aspects of these limits are also implementation-defined. Some implementations may use dynamic allocation techniques and have no specific limit values to report.

The format of the **y.output** file is not given because specification of the format was not seen to enhance applications portability. The listing is primarily intended to help human users understand and debug the parser; use of **y.output** by a conforming application script would be unusual. Furthermore, implementations have not produced consistent output and no popular format was apparent. The format selected by the implementation should be human-readable, in addition to the requirement that it be a text file.

Standard error reports are not specifically described because they are seldom of use to conforming applications and there was no reason to restrict implementations.

Some implementations recognize `"={"` as equivalent to `'{'` because it appears in historical documentation. This construction was recognized and documented as obsolete as long ago as 1978, in the referenced *Yacc: Yet Another Compiler-Compiler*. This volume of IEEE Std 1003.1-200x chose to leave it as obsolete and omit it.

Multi-byte characters should be recognized by the lexical analyzer and returned as tokens. They should not be returned as multi-byte character literals. The token **error** that is used for error recovery is normally assigned the value 256 in the historical implementation. Thus, the token value 256, which used in many multi-byte character sets, is not available for use as the value of a user-defined token.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *lex*

CHANGE HISTORY

First released in Issue 2.

Issue 5

FUTURE DIRECTIONS section added.

Issue 6

This utility is now marked as part of the C-Language Development Utilities option.

Minor changes have been added to align with the IEEE P1003.2b draft standard.

The normative text is reworded to avoid use of the term ‘‘must’’ for application requirements.

IEEE PASC Interpretation 1003.2 #177 is applied, changing the comment on **RCURL** from the **}%** token to the **%}**.

NAME

zcat — expand and concatenate data

SYNOPSIS

XSI `zcat [file...]`

DESCRIPTION

The *zcat* utility shall write to standard output the uncompressed form of files that have been compressed using the *compress* utility. It is the equivalent of *uncompress* **–c**. Input files are not affected.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

file	The pathname of a file previously processed by the *compress* utility. If *file* already has the **.Z** suffix specified, it is used as submitted. Otherwise, the **.Z** suffix is appended to the filename prior to processing.

STDIN

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `'–'`.

INPUT FILES

Input files shall be compressed files that are in the format produced by the *compress* utility.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *zcat*:

LANG	Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-200x, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)
LC_ALL	If set to a non-empty string value, override the values of all the other internationalization variables.
LC_CTYPE	Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).
LC_MESSAGES	Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.
NLSPATH	Determine the location of message catalogs for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The compressed files given as input shall be written on standard output in their uncompressed form.

STDERR

The standard error shall be used only for diagnostic messages. |

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

compress, *uncompress*

CHANGE HISTORY

First released in Issue 4.

Index

Rationale (Informative)

Part A:

Base Definitions

The Open Group
The Institute of Electrical and Electronics Engineers, Inc.

6

Appendix A

Rationale for Base Definitions

A.1 Introduction

A.1.1 Scope

IEEE Std 1003.1-200x is one of a family of standards known as POSIX. The family of standards extends to many topics; IEEE Std 1003.1-200x is known as POSIX.1 and consists of both operating system interfaces and shell and utilities. IEEE Std 1003.1-200x is technically identical to The Open Group Base Specifications, Issue 6, which comprise the core volumes of the Single UNIX Specification, Version 3.

Scope of IEEE Std 1003.1-200x

The (paraphrased) goals of this development were to produce a single common revision to the overlapping POSIX.1 and POSIX.2 standards, and the Single UNIX Specification, Version 2. As such, the scope of the revision includes the scopes of the original documents merged.

Since the revision includes merging the Base volumes of the Single UNIX Specification, many features that were previously not *adopted* into earlier revisions of POSIX.1 and POSIX.2 are now included in IEEE Std 1003.1-200x. In most cases, these additions are part of the XSI extension; in other cases the standard developers decided that now was the time to migrate these to the base standard.

The Single UNIX Specification programming environment provides a broad-based functional set of interfaces to support the porting of existing UNIX applications and the development of new applications. The environment also supports a rich set of tools for application development.

The majority of the obsolescent material from the existing POSIX.1 and POSIX.2 standards, and material marked LEGACY from The Open Group's Base specifications, has been removed in this revision. New members of the Legacy Option Group have been added, reflecting the advance in understanding of what is required.

The following IEEE Standards have been added to the base documents in this revision:

- IEEE Std 1003.1d-1999
- IEEE Std 1003.1j-2000
- IEEE Std 1003.1q-2000
- IEEE P1003.1a draft standard
- IEEE Std 1003.2d-1994
- IEEE P1003.2b draft standard
- Selected parts of IEEE Std 1003.1g-2000

Only selected parts of IEEE Std 1003.1g-2000 were included. This was because there is much duplication between the XNS, Issue 5.2 specification (another base document) and the material from IEEE Std 1003.1g-2000, the former document being aligned with the latest networking specifications for IPv6. Only the following sections of IEEE Std 1003.1g-2000 were considered for inclusion:

- General terms related to sockets (2.2.2)
- Socket concepts (5.1 through 5.3 inclusive)
- The *pselect*( ) function (6.2.2.1 and 6.2.3)
- The **<sys/select.h>** header (6.2)

The following were requirements on IEEE Std 1003.1-200x:

- Backward-compatibility

  It was agreed that there should be no breakage of functionality in the existing base documents. This requirement was tempered by changes introduced due to interpretations and corrigenda on the base documents, and any changes introduced in the ISO / IEC 9899: 1999 standard (C Language).

- Architecture and n-bit neutral

  The common standard should not make any implicit assumptions about the system architecture or size of data types; for example, previously some 32-bit implicit assumptions had crept into the standards.

- Extensibility

  It should be possible to extend the common standard without breaking backward-compatibility. For example, the name space should be reserved and structured to avoid duplication of names between the standard and extensions to it.

POSIX.1 and the ISO C standard

Previous revisions of POSIX.1 built upon the ISO C standard by reference only. This revision takes a different approach.

The standard developers believed it essential for a programmer to have a single complete reference place, but recognized that deference to the formal standard had to be addressed for the duplicate interface definitions between the ISO C standard and the Single UNIX Specification.

It was agreed that where an interface has a version in the ISO C standard, the DESCRIPTION section should describe the relationship to the ISO C standard and markings should be added as appropriate to show where the ISO C standard has been extended in the text.

The following block of text was added to each reference page affected:

> *The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-200x defers to the ISO C standard.*

and each page was parsed for additions beyond the ISO C standard (that is, including both POSIX and UNIX extensions), and these extensions are marked as CX extensions (for C Extensions).

FIPS Requirements

The Federal Information Processing Standards (FIPS) are a series of U.S. government procurement standards managed and maintained on behalf of the U.S. Department of Commerce by the National Institute of Standards and Technology (NIST).

The following restrictions have been made in this version of IEEE Std 1003.1 in order to align with FIPS 151-2 requirements:

- The implementation supports _POSIX_CHOWN_RESTRICTED.
- The limit {NGROUPS_MAX} is now greater than or equal to 8.
- The implementation supports the setting of the group ID of a file (when it is created) to that of the parent directory.
- The implementation supports _POSIX_SAVED_IDS.
- The implementation supports _POSIX_VDISABLE.
- The implementation supports _POSIX_JOB_CONTROL.
- The implementation supports _POSIX_NO_TRUNC.
- The *read*( ) function returns the number of bytes read when interrupted by a signal and does not return –1.
- The *write*( ) function returns the number of bytes written when interrupted by a signal and does not return –1.
- In the environment for the login shell, the environment variables *LOGNAME* and *HOME* are defined and have the properties described in IEEE Std 1003.1-200x.
- The value of {CHILD_MAX} is now greater than or equal to 25.
- The value of {OPEN_MAX} is now greater than or equal to 20.
- The implementation supports the functionality associated with the symbols CS7, CS8, CSTOPB, PARODD, and PARENB defined in **<termios.h>**.

A.1.2 Conformance

See Section A.2 (on page 3299).

A.1.3 Normative References

There is no additional rationale provided for this section.

A.1.4 Terminology

The meanings specified in IEEE Std 1003.1-200x for the words *shall*, *should*, and *may* are mandated by ISO/IEC directives.

In the Rationale (Informative) volume of IEEE Std 1003.1-200x, the words *shall*, *should*, and *may* are sometimes used to illustrate similar usages in IEEE Std 1003.1-200x. However, the rationale itself does not specify anything regarding implementations or applications.

conformance document

As a practical matter, the conformance document is effectively part of the system documentation. Conformance documents are distinguished by IEEE Std 1003.1-200x so that they can be referred to distinctly.

implementation-defined

This definition is analogous to that of the ISO C standard and, together with *undefined* and *unspecified*, provides a range of specification of freedom allowed to the interface implementor.

may

The use of *may* has been limited as much as possible, due both to confusion stemming from its ordinary English meaning and to objections regarding the desirability of having as few options as possible and those as clearly specified as possible.

The usage of *can* and *may* were selected to contrast optional application behavior (can) against optional implementation behavior (may).

shall

Declarative sentences are sometimes used in IEEE Std 1003.1-200x as if they included the word *shall*, and facilities thus specified are no less required. For example, the two statements:

1. The *foo*( ) function shall return zero.
2. The *foo*( ) function returns zero.

are meant to be exactly equivalent.

should

In IEEE Std 1003.1-200x, the word *should* does not usually apply to the implementation, but rather to the application. Thus, the important words regarding implementations are *shall*, which indicates requirements, and *may*, which indicates options.

obsolescent

The term *obsolescent* means ‘‘do not use this feature in new applications’’. The obsolescence concept is not an ideal solution, but was used as a method of increasing consensus: many more objections would be heard from the user community if some of these historical features were suddenly withdrawn without the grace period obsolescence implies. The phrase ‘‘may be considered for withdrawal in future revisions’’ implies that the result of that consideration might in fact keep those features indefinitely if the predominance of applications do not migrate away from them quickly.

legacy

The term *legacy* was added for compatibility with the Single UNIX Specification. It means ‘‘this feature is historic and optional; do not use this feature in new applications. There are alternate interfaces that are more suitable.’’. It is used exclusively for XSI extensions, and includes facilities that were mandatory in previous versions of the base document but are optional in this revision. This is a way to ‘‘sunset’’ the usage of certain functions. Application writers should not rely on the existence of these facilities in new applications, but should follow the migration path detailed in the APPLICATION USAGE sections of the relevant pages.

The terms *legacy* and *obsolescent* are different: a feature marked LEGACY is not recommended for new work and need not be present on an implementation (if the XSI Legacy Option Group is not supported). A feature noted as obsolescent is supported by all implementations, but may be removed in a future revision; new applications should not use these features.

system documentation

The system documentation should normally describe the whole of the implementation, including any extensions provided by the implementation. Such documents normally contain information at least as detailed as the specifications in IEEE Std 1003.1-200x. Few requirements are made on the system documentation, but the term is needed to avoid a dangling pointer where the conformance document is permitted to point to the system documentation.

undefined

See *implementation-defined*.

unspecified

See *implementation-defined*.

The definitions for *unspecified* and *undefined* appear nearly identical at first examination, but are not. The term *unspecified* means that a conforming program may deal with the unspecified behavior, and it should not care what the outcome is. The term *undefined* says that a conforming program should not do it because no definition is provided for what it does (and implicitly it would care what the outcome was if it tried it). It is important to remember, however, that if the syntax permits the statement at all, it must have some outcome in a real implementation.

Thus, the terms *undefined* and *unspecified* apply to the way the application should think about the feature. In terms of the implementation, it is always ‘‘defined’’—there is always some result, even if it is an error. The implementation is free to choose the behavior it prefers.

This also implies that an implementation, or another standard, could specify or define the result in a useful fashion. The terms apply to IEEE Std 1003.1-200x specifically.

The term *implementation-defined* implies requirements for documentation that are not required for *undefined* (or *unspecified*). Where there is no need for a conforming program to know the definition, the term *undefined* is used, even though *implementation-defined* could also have been used in this context. There could be a fourth term, specifying ‘‘this standard does not say what this does; it is acceptable to define it in an implementation, but it does not need to be documented’’, and undefined would then be used very rarely for the few things for which any definition is not useful. In particular, implementation-defined is used where it is believed that certain classes of application will need to know such details to determine whether the application can be successfully ported to the implementation. Such applications are not always strictly portable, but nevertheless are common and useful; often the requirements met by the application cannot be met without dealing with the issues implied by ‘‘implementation-defined’’.

In many places IEEE Std 1003.1-200x is silent about the behavior of some possible construct. For example, a variable may be defined for a specified range of values and behaviors are described for those values; nothing is said about what happens if the variable has any other value. That kind of silence can imply an error in the standard, but it may also imply that the standard was intentionally silent and that any behavior is permitted. There is a natural tendency to infer that if the standard is silent, a behavior is prohibited. That is not the intent. Silence is intended to be equivalent to the term *unspecified*.

The term *application* is not defined in IEEE Std 1003.1-200x; it is assumed to be a part of general computer science terminology.

Three terms used within IEEE Std 1003.1-200x overlap in meaning: ‘‘macro’’, ‘‘symbolic name’’, and ‘‘symbolic constant’’.

macro

This usually describes a C preprocessor symbol, the result of the **#define** operator, with or without an argument. It may also be used to describe similar mechanisms in editors and text processors.

symbolic name

This can also refer to a C preprocessor symbol (without arguments), but is also used to refer to the names for characters in character sets. In addition, it is sometimes used to refer to host names and even filenames.

symbolic constant

This also refers to a C preprocessor symbol (also without arguments).

In most cases, the difference in semantic content is negligible to nonexistent. Readers should not attempt to read any meaning into the various usages of these terms.

A.1.5 Portability

To aid the identification of options within IEEE Std 1003.1-200x, a notation consisting of margin codes and shading is used. This is based on the notation used in previous revisions of The Open Group's Base specifications.

The benefits of this approach is a reduction in the number of *if* statements within the running text, that makes the text easier to read, and also an identification to the programmer that they need to ensure that their target platforms support the underlying options. For example, if functionality is marked with THR in the margin, it will be available on all systems supporting the Threads option, but may not be available on some others.

A.1.5.1 Codes

This section includes codes for options defined in the Base Definitions volume of IEEE Std 1003.1-200x, Section 2.1.6, Options, and the following additional codes for other purposes:

CX	This margin code is used to denote extensions beyond the ISO C standard. For interfaces that are duplicated between IEEE Std 1003.1-200x and the ISO C standard, a CX introduction block describes the nature of the duplication, with any extensions appropriately CX marked and shaded. Where an interface is added to an ISO C standard header, within the header the interface has an appropriate margin marker and shading (for example, CX, XSI, TSF, and so on) and the same marking appears on the reference page in the SYNOPSIS section. This enables a programmer to easily identify that the interface is extending an ISO C standard header.
MX	This margin code is used to denote IEC 60559:1989 standard floating-point extensions.
OB	This margin code is used to denote obsolescent behavior and thus flag a possible future application portability warning.
OH	The Single UNIX Specification has historically tried to reduce the number of headers an application has had to include when using a particular interface. Sometimes this was fewer than the base standard, and hence a notation is used to flag which headers are optional if you are using a system supporting the XSI extension.
XSI	This code is used to denote interfaces and facilities within interfaces only required on systems supporting the XSI extension. This is introduced to support the Single UNIX Specification.
XSR	This code is used to denote interfaces and facilities within interfaces only required on systems supporting STREAMS. This is introduced to support the Single UNIX Specification, although it is defined in a way so that it can standalone from the XSI notation.

A.1.5.2 Margin Code Notation

Since some features may depend on one or more options, or require more than one options, a notation is used. Where a feature requires support of a single option, a single margin code will occur in the margin. If it depends on two options and both are required, then the codes will appear with a <space> separator. If either of two options are required then a logical OR is denoted using the **'|'** symbol. If more than two codes are used, a special notation is used.

A.2 Conformance

The terms *profile* and *profiling* are used throughout this section.

A profile of a standard or standards is a codified set of option selections, such that by being conformant to a profile, particular classes of users are specifically supported.

These conformance definitions are descended from those in the ISO POSIX-1:1996 standard, but with changes for the following:

- The addition of profiling options, allowing larger profiles of options such as the XSI extension used by the Single UNIX Specification. In effect, it has profiled itself (that is, created a self-profile).
- The addition of a definition of subprofiling considerations, to allow smaller profiles of options.
- The addition of a hierarchy of super-options for XSI; these were formerly known as *Feature Groups* in The Open Group System Interfaces and Headers, Issue 5 specification.
- Options from the ISO POSIX-2:1993 standard are also now included as IEEE Std 1003.1-200x merges the functionality from it.

A.2.1 Implementation Conformance

These definitions allow application developers to know what to depend on in an implementation.

There is no definition of a *strictly conforming implementation*; that would be an implementation that provides *only* those facilities specified by POSIX.1 with no extensions whatsoever. This is because no actual operating system implementation can exist without system administration and initialization facilities that are beyond the scope of POSIX.1.

A.2.1.1 Requirements

The word ''support'' is used in certain instances, rather than ''provide'', in order to allow an implementation that has no resident software development facilities, but that supports the execution of a *Strictly Conforming POSIX.1 Application*, to be a *conforming implementation*.

A.2.1.2 Documentation

The conformance documentation is required to use the same numbering scheme as POSIX.1 for purposes of cross-referencing. All options that an implementation chooses shall be reflected in **<limits.h>** and **<unistd.h>**.

Note that the use of ''may'' in terms of where conformance documents record where implementations may vary, implies that it is not required to describe those features identified as undefined or unspecified.

Other aspects of systems must be evaluated by purchasers for suitability. Many systems incorporate buffering facilities, maintaining updated data in volatile storage and transferring such updates to non-volatile storage asynchronously. Various exception conditions, such as a power failure or a system crash, can cause this data to be lost. The data may be associated with a file that is still open, with one that has been closed, with a directory, or with any other internal system data structures associated with permanent storage. This data can be lost, in whole or part, so that only careful inspection of file contents could determine that an update did not occur.

Also, interrelated file activities, where multiple files and/or directories are updated, or where space is allocated or released in the file system structures, can leave inconsistencies in the relationship between data in the various files and directories, or in the file system itself. Such inconsistencies can break applications that expect updates to occur in a specific sequence, so that updates in one place correspond with related updates in another place.

For example, if a user creates a file, places information in the file, and then records this action in another file, a system or power failure at this point followed by restart may result in a state in which the record of the action is permanently recorded, but the file created (or some of its information) has been lost. The consequences of this to the user may be undesirable. For a user on such a system, the only safe action may be to require the system administrator to have a policy that requires, after any system or power failure, that the entire file system must be restored from the most recent backup copy (causing all intervening work to be lost).

The characteristics of each implementation will vary in this respect and may or may not meet the requirements of a given application or user. Enforcement of such requirements is beyond the scope of POSIX.1. It is up to the purchaser to determine what facilities are provided in an implementation that affect the exposure to possible data or sequence loss, and also what underlying implementation techniques and/or facilities are provided that reduce or limit such loss or its consequences.

A.2.1.3 POSIX Conformance

This really means conformance to the base standard; however, since this revision includes the core material of the Single UNIX Specification, the standard developers decided that it was appropriate to segment the conformance requirements into two, the former for the base standard, and the latter for the Single UNIX Specification.

Within POSIX.1 there are some symbolic constants that, if defined, indicate that a certain option is enabled. Other symbolic constants exist in POSIX.1 for other reasons.

As part of the revision some alignment has occurred of the options with the FIPS 151-2 profile on the POSIX.1-1990 standard. The following options from the POSIX.1-1990 standard are now mandatory:

- _POSIX_JOB_CONTROL
- _POSIX_SAVED_IDS
- _POSIX_VDISABLE

A POSIX-conformant system may support the XSI extensions of the Single UNIX Specification. This was intentional since the standard developers intend them to be upwards-compatible, so that a system conforming to the Single UNIX Specification can also conform to the base standard at the same time.

A.2.1.4 XSI Conformance

This section is added since the revision merges in the base volumes of the Single UNIX Specification.

XSI conformance can be thought of as a profile, selecting certain options from IEEE Std 1003.1-200x.

A.2.1.5 Option Groups

The concept of *Option Groups* is introduced to IEEE Std 1003.1-200x to allow collections of related functions or options to be grouped together. This has been used as follows: the *XSI Option Groups* have been created to allow super-options, collections of underlying options and related functions, to be collectively supported by XSI-conforming systems. These reflect the *Feature Groups* from The Open Group System Interfaces and Headers, Issue 5 specification.

The standard developers considered the matter of subprofiling and decided it was better to include an enabling mechanism rather than detailed normative requirements. A set of subprofiling options was developed and included later in this volume of IEEE Std 1003.1-200x as an informative illustration.

A.2.1.6 Options

The final subsections within *Implementation Conformance* list the core options within IEEE Std 1003.1-200x. This includes both options for the System Interfaces volume of IEEE Std 1003.1-200x and the Shell and Utilities volume of IEEE Std 1003.1-200x.

A.2.2 Application Conformance

These definitions guide users or adaptors of applications in determining on which implementations an application will run and how much adaptation would be required to make it run on others. These definitions are modeled after related ones in the ISO C standard.

POSIX.1 occasionally uses the expressions *portable application* or *conforming application*. As they are used, these are synonyms for any of these terms. The differences between the classes of application conformance relate to the requirements for other standards, the options supported (such as the XSI extension) or, in the case of the Conforming POSIX.1 Application Using Extensions, to implementation extensions. When one of the less explicit expressions is used, it should be apparent from the context of the discussion which of the more explicit names is appropriate

A.2.2.1 Strictly Conforming POSIX Application

This definition is analogous to that of a ISO C standard *conforming program*.

The major difference between a *Strictly Conforming POSIX Application* and a ISO C standard *strictly conforming program* is that the latter is not allowed to use features of POSIX that are not in the ISO C standard.

A.2.2.2 Conforming POSIX Application

Examples of <National Bodies> include ANSI, BSI, and AFNOR.

A.2.2.3 Conforming POSIX Application Using Extensions

Due to possible requirements for configuration or implementation characteristics in excess of the specifications in **<limits.h>** or related to the hardware (such as array size or file space), not every Conforming POSIX Application Using Extensions will run on every conforming implementation.

A.2.2.4 Strictly Conforming XSI Application

This is intended to be upwards-compatible with the definition of a Strictly Conforming POSIX Application, with the addition of the facilities and functionality included in the XSI extension.

A.2.2.5 Conforming XSI Application Using Extensions

Such applications may use extensions beyond the facilities defined by IEEE Std 1003.1-200x including the XSI extension, but need to document the additional requirements.

A.2.3 Language-Dependent Services for the C Programming Language

POSIX.1 is, for historical reasons, both a specification of an operating system interface, shell and utilities, and a C binding for that specification. Efforts had been previously undertaken to generate a language-independent specification; however, that had failed, and the fact that the ISO C standard is the *de facto* primary language on POSIX and the UNIX system makes this a necessary and workable situation.

A.2.4 Other Language-Related Specifications

There is no additional rationale provided for this section.

A.3 Definitions

The definitions in this section are stated so that they can be used as exact substitutes for the terms in text. They should not contain requirements or cross-references to sections within IEEE Std 1003.1-200x; that is accomplished by using an informative note. In addition, the term should not be included in its own definition. Where requirements or descriptions need to be addressed but cannot be included in the definitions, due to not meeting the above criteria, these occur in the General Concepts chapter.

In this revision, the definitions have been reworked extensively to meet style requirements and to include terms from the base documents (see the Scope).

Many of these definitions are necessarily circular, and some of the terms (such as *process*) are variants of basic computing science terms that are inherently hard to define. Where some definitions are more conceptual and contain requirements, these appear in the General Concepts chapter. Those listed in this section appear in an alphabetical glossary format of terms.

Some definitions must allow extension to cover terms or facilities that are not explicitly mentioned in IEEE Std 1003.1-200x. For example, the definition of *Extended Security Controls* permits implementations beyond those defined in IEEE Std 1003.1-200x.

Some terms in the following list of notes do not appear in IEEE Std 1003.1-200x; these are marked prefixed with a asterisk (*). Many of them have been specifically excluded from IEEE Std 1003.1-200x because they concern system administration, implementation, or other issues that are not specific to the programming interface. Those are marked with a reason, such as ‘‘implementation-defined’’.

Appropriate Privileges

One of the fundamental security problems with many historical UNIX systems has been that the privilege mechanism is monolithic—a user has either no privileges or *all* privileges. Thus, a successful ‘‘trojan horse’’ attack on a privileged process defeats all security provisions. Therefore, POSIX.1 allows more granular privilege mechanisms to be defined. For many historical implementations of the UNIX system, the presence of the term *appropriate privileges* in POSIX.1 may be understood as a synonym for *superuser* (UID 0). However, other systems have emerged where this is not the case and each discrete controllable action has *appropriate privileges* associated with it. Because this mechanism is implementation-defined, it must be described in the conformance document. Although that description affects several parts of POSIX.1 where the term *appropriate privilege* is used, because the term *implementation-defined* only appears here, the description of the entire mechanism and its effects on these other sections belongs in this equivalent section of the conformance document. This is especially convenient for implementations with a single mechanism that applies in all areas, since it only needs to be described once.

Byte

The restriction that a byte is now exactly eight bits was a conscious decision by the standard developers. It came about due to a combination of factors, primarily the use of the type **int8_t** within the networking functions and the alignment with the ISO / IEC 9899: 1999 standard, where the **intN_t** types are now defined.

According to the ISO / IEC 9899: 1999 standard:

- The **[u]intN_t** types must be two’s complement with no padding bits and no illegal values.
- All types (apart from bit fields, which are not relevant here) must oocupy an integral number of bytes.
- If a type with width W occupies B bytes with C bits per byte (C is the value of {CHAR_BIT}), then it has P padding bits where $P+W=B*C$.
- For **int8_t** we therefore have $P=0$, $W=8$. Since $B\geq1$, $C\geq8$, the only solution is $B=1$, $C=8$.

The standard developers also felt that this was not an undue restriction for the current state of the art for this version of IEEE Std 1003.1-200x, but recognize that if industry trends continue, a wider character type may be required in the future.

Character

The term *character* is used to mean a sequence of one or more bytes representing a single graphic symbol. The deviation in the exact text of the ISO C standard definition for *byte* meets the intent of the rationale of the ISO C standard also clears up the ambiguity raised by the term *basic execution character set*. The octet-minimum requirement is a reflection of the {CHAR_BIT} value.

Clock Tick

The ISO C standard defines a similar interval for use by the *clock*( ) function. There is no requirement that these intervals be the same. In historical implementations these intervals are different.

Command

The terms *command* and *utility* are related but have distinct meanings. Command is defined as ‘‘a directive to a shell to perform a specific task’’. The directive can be in the form of a single utility name (for example, *ls*), or the directive can take the form of a compound command (for example, `"ls | grep name | pr"`). A utility is a program that can be called by name from a shell. Issuing only the name of the utility to a shell is the equivalent of a one-word command. A utility may be invoked as a separate program that executes in a different process than the command language interpreter, or it may be implemented as a part of the command language interpreter. For example, the *echo* command (the directive to perform a specific task) may be implemented such that the *echo* utility (the logic that performs the task of echoing) is in a separate program; therefore, it is executed in a process that is different from the command language interpreter. Conversely, the logic that performs the *echo* utility could be built into the command language interpreter; therefore, it could execute in the same process as the command language interpreter.

The terms *tool* and *application* can be thought of as being synonymous with *utility* from the perspective of the operating system kernel. Tools, applications, and utilities historically have run, typically, in processes above the kernel level. Tools and utilities historically have been a part of the operating system non-kernel code and have performed system-related functions, such as listing directory contents, checking file systems, repairing file systems, or extracting system status information. Applications have not generally been a part of the operating system, and they perform non-system-related functions, such as word processing, architectural design, mechanical design, workstation publishing, or financial analysis. Utilities have most frequently been provided by the operating system distributor, applications by third-party software distributors, or by the users themselves. Nevertheless, IEEE Std 1003.1-200x does not differentiate between tools, utilities, and applications when it comes to receiving services from the system, a shell, or the standard utilities. (For example, the *xargs* utility invokes another utility; it would be of fairly limited usefulness if the users could not run their own applications in place of the standard utilities.) Utilities are not applications in the sense that they are not themselves subject to the restrictions of IEEE Std 1003.1-200x or any other standard—there is no requirement for *grep*, *stty*, or any of the utilities defined here to be any of the classes of conforming applications.

Column Positions

In most 1-byte character sets, such as ASCII, the concept of column positions is identical to character positions and to bytes. Therefore, it has been historically acceptable for some implementations to describe line folding or tab stops or table column alignment in terms of bytes or character positions. Other character sets pose complications, as they can have internal representations longer than one octet and they can have display characters that have different widths on the terminal screen or printer.

In IEEE Std 1003.1-200x the term *column positions* has been defined to mean character—not byte—positions in input files (such as ‘‘column position 7 of the FORTRAN input’’). Output files describe the column position in terms of the display width of the narrowest printable character in the character set, adjusted to fit the characteristics of the output device. It is very possible that *n* column positions will not be able to hold *n* characters in some character sets, unless all of those characters are of the narrowest width. It is assumed that the implementation is aware of the width of the various characters, deriving this information from the value of *LC_CTYPE*, and thus can determine how many column positions to allot for each character in those utilities where it is important.

The term *column position* was used instead of the more natural *column* because the latter is frequently used in the different contexts of columns of figures, columns of table values, and so on. Wherever confusion might result, these latter types of columns are referred to as *text* *columns*.

Controlling Terminal

The question of which of possibly several special files referring to the terminal is meant is not addressed in POSIX.1. The filename **/dev/tty** is a synonym for the controlling terminal associated with a process.

Device Number*

The concept is handled in *stat*( ) as *ID of device*.

Direct I/O

Historically, direct I/O refers to the system bypassing intermediate buffering, but may be extended to cover implementation-defined optimizations.

Directory

The format of the directory file is implementation-defined and differs radically between System V and 4.3 BSD. However, routines (derived from 4.3 BSD) for accessing directories and certain constraints on the format of the information returned by those routines are described in the **<dirent.h>** header.

Directory Entry

Throughout IEEE Std 1003.1-200x, the term *link* is used (about the *link*( ) function, for example) in describing the objects that point to files from directories.

Display

The Shell and Utilities volume of IEEE Std 1003.1-200x assigns precise requirements for the terms *display* and *write*. Some historical systems have chosen to implement certain utilities without using the traditional file descriptor model. For example, the *vi* editor might employ direct screen memory updates on a personal computer, rather than a *write*( ) system call. An instance of user prompting might appear in a dialog box, rather than with standard error. When the Shell and Utilities volume of IEEE Std 1003.1-200x uses the term *display*, the method of outputting to the terminal is unspecified; many historical implementations use *termcap* or *terminfo*, but this is not a requirement. The term *write* is used when the Shell and Utilities volume of IEEE Std 1003.1-200x mandates that a file descriptor be used and that the output can be redirected. However, it is assumed that when the writing is directly to the terminal (it has not been redirected elsewhere), there is no practical way for a user or test suite to determine whether a file descriptor is being used. Therefore, the use of a file descriptor is mandated only for the redirection case and the implementation is free to use any method when the output is not redirected. The verb *write* is used almost exclusively, with the very few exceptions of those utilities where output redirection need not be supported: *tabs*, *talk*, *tput*, and *vi*.

Dot

The symbolic name *dot* is carefully used in POSIX.1 to distinguish the working directory filename from a period or a decimal point.

Dot-Dot

Historical implementations permit the use of these filenames without their special meanings. Such use precludes any meaningful use of these filenames by a Conforming POSIX.1 Application. Therefore, such use is considered an extension, the use of which makes an implementation non-conforming; see also Section A.4.11 (on page 3327).

Epoch

Historically, the origin of UNIX system time was referred to as ‘‘00:00:00 GMT, January 1, 1970’’. Greenwich Mean Time is actually not a term acknowledged by the international standards community; therefore, this term, *Epoch*, is used to abbreviate the reference to the actual standard, Coordinated Universal Time.

FIFO Special File

See *pipe* in **Pipe** (on page 3314).

File

It is permissible for an implementation-defined file type to be non-readable or non-writable.

File Classes

These classes correspond to the historical sets of permission bits. The classes are general to allow implementations flexibility in expanding the access mechanism for more stringent security environments. Note that a process is in one and only one class, so there is no ambiguity.

Filename

At the present time, the primary responsibility for truncating filenames containing multi-byte characters must reside with the application. Some industry groups involved in internationalization believe that in the future the responsibility must reside with the kernel. For the moment, a clearer understanding of the implications of making the kernel responsible for truncation of multi-byte filenames is needed.

Character-level truncation was not adopted because there is no support in POSIX.1 that advises how the kernel distinguishes between single and multi-byte characters. Until that time, it must be incumbent upon application writers to determine where multi-byte characters must be truncated.

File System

Historically, the meaning of this term has been overloaded with two meanings: that of the complete file hierarchy, and that of a mountable subset of that hierarchy; that is, a mounted file system. POSIX.1 uses the term *file system* in the second sense, except that it is limited to the scope of a process (and a process’ root directory). This usage also clarifies the domain in which a file serial number is unique.

Graphic Character

This definition is made available for those definitions (in particular, *TZ*) which must exclude control characters.

Group Database

See **User Database** (on page 3322).

Group File*

Implementation-defined; see **User Database** (on page 3322).

Historical Implementations*

This refers to previously existing implementations of programming interfaces and operating systems that are related to the interface specified by POSIX.1.

Hosted Implementation*

This refers to a POSIX.1 implementation that is accomplished through interfaces from the POSIX.1 services to some alternate form of operating system kernel services. Note that the line between a hosted implementation and a native implementation is blurred, since most implementations will provide some services directly from the kernel and others through some indirect path. (For example, *fopen*( ) might use *open*( ); or *mkfifo*( ) might use *mknod*( ).) There is no necessary relationship between the type of implementation and its correctness, performance, and/or reliability.

Implementation*

This term is generally used instead of its synonym, *system*, to emphasize the consequences of decisions to be made by system implementors. Perhaps if no options or extensions to POSIX.1 were allowed, this usage would not have occurred.

The term *specific implementation* is sometimes used as a synonym for *implementation*. This should not be interpreted too narrowly; both terms can represent a relatively broad group of systems. For example, a hardware vendor could market a very wide selection of systems that all used the same instruction set, with some systems desktop models and others large multi-user minicomputers. This wide range would probably share a common POSIX.1 operating system, allowing an application compiled for one to be used on any of the others; this is a [*specific*]*implementation*.

However, that wide range of machines probably has some differences between the models. Some may have different clock rates, different file systems, different resource limits, different network connections, and so on, depending on their sizes or intended usages. Even on two identical machines, the system administrators may configure them differently. Each of these different systems is known by the term *a specific instance of a specific implementation*. This term is only used in the portions of POSIX.1 dealing with runtime queries: *sysconf*( ) and *pathconf*( ).

Incomplete Pathname*

Absolute pathname has been adequately defined.

Job Control

In order to understand the job control facilities in POSIX.1 it is useful to understand how they are used by a job control-cognizant shell to create the user interface effect of job control.

While the job control facilities supplied by POSIX.1 can, in theory, support different types of interactive job control interfaces supplied by different types of shells, there was historically one | particular interface that was most common when the standard was originally developed | (provided by BSD C Shell). This discussion describes that interface as a means of illustrating how the POSIX.1 job control facilities can be used.

Job control allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter (shell).

The user can launch jobs (command pipelines) in either the foreground or background. When launched in the foreground, the shell waits for the job to complete before prompting for additional commands. When launched in the background, the shell does not wait, but immediately prompts for new commands.

If the user launches a job in the foreground and subsequently regrets this, the user can type the suspend character (typically set to <control>-Z), which causes the foreground job to stop and the shell to begin prompting for new commands. The stopped job can be continued by the user (via special shell commands) either as a foreground job or as a background job. Background jobs can also be moved into the foreground via shell commands.

If a background job attempts to access the login terminal (controlling terminal), it is stopped by the terminal driver and the shell is notified, which, in turn, notifies the user. (Terminal access includes *read*( ) and certain terminal control functions, and conditionally includes *write*( ).) The user can continue the stopped job in the foreground, thus allowing the terminal access to succeed in an orderly fashion. After the terminal access succeeds, the user can optionally move the job into the background via the suspend character and shell commands.

Implementing Job Control Shells

The interactive interface described previously can be accomplished using the POSIX.1 job control facilities in the following way.

The key feature necessary to provide job control is a way to group processes into jobs. This grouping is necessary in order to direct signals to a single job and also to identify which job is in the foreground. (There is at most one job that is in the foreground on any controlling terminal at a time.)

The concept of *process groups* is used to provide this grouping. The shell places each job in a separate process group via the *setpgid*( ) function. To do this, the *setpgid*( ) function is invoked by the shell for each process in the job. It is actually useful to invoke *setpgid*( ) twice for each process: once in the child process, after calling *fork*( ) to create the process, but before calling one of the *exec* family of functions to begin execution of the program, and once in the parent shell process, after calling *fork*( ) to create the child. The redundant invocation avoids a race condition by ensuring that the child process is placed into the new process group before either the parent or the child relies on this being the case. The *process group ID* for the job is selected by the shell to be equal to the *process ID* of one of the processes in the job. Some shells choose to make one process in the job be the parent of the other processes in the job (if any). Other shells (for example, the C Shell) choose to make themselves the parent of all processes in the pipeline (job). In order to support this latter case, the *setpgid*( ) function accepts a process group ID parameter since the correct process group ID cannot be inherited from the shell. The shell itself is considered to be a job and is the sole process in its own process group.

The shell also controls which job is currently in the foreground. A foreground and background job differ in two ways: the shell waits for a foreground command to complete (or stop) before continuing to read new commands, and the terminal I/O driver inhibits terminal access by background jobs (causing the processes to stop). Thus, the shell must work cooperatively with the terminal I/O driver and have a common understanding of which job is currently in the foreground. It is the user who decides which command should be currently in the foreground, and the user informs the shell via shell commands. The shell, in turn, informs the terminal I/O driver via the *tcsetpgrp*( ) function. This indicates to the terminal I/O driver the process group ID of the foreground process group (job). When the current foreground job either stops or terminates, the shell places itself in the foreground via *tcsetpgrp*( ) before prompting for additional commands. Note that when a job is created the new process group begins as a background process group. It requires an explicit act of the shell via *tcsetpgrp*( ) to move a process group (job) into the foreground.

When a process in a job stops or terminates, its parent (for example, the shell) receives synchronous notification by calling the *waitpid*( ) function with the WUNTRACED flag set. Asynchronous notification is also provided when the parent establishes a signal handler for SIGCHLD and does not specify the SA_NOCLDSTOP flag. Usually all processes in a job stop as a unit since the terminal I/O driver always sends job control stop signals to all processes in the process group.

To continue a stopped job, the shell sends the SIGCONT signal to the process group of the job. In addition, if the job is being continued in the foreground, the shell invokes *tcsetpgrp*( ) to place the job in the foreground before sending SIGCONT. Otherwise, the shell leaves itself in the foreground and reads additional commands.

There is additional flexibility in the POSIX.1 job control facilities that allows deviations from the typical interface. Clearing the TOSTOP terminal flag allows background jobs to perform *write*( ) functions without stopping. The same effect can be achieved on a per-process basis by having a process set the signal action for SIGTTOU to SIG_IGN.

Note that the terms *job* and *process group* can be used interchangeably. A login session that is not using the job control facilities can be thought of as a large collection of processes that are all in the same job (process group). Such a login session may have a partial distinction between foreground and background processes; that is, the shell may choose to wait for some processes before continuing to read new commands and may not wait for other processes. However, the terminal I/O driver will consider all these processes to be in the foreground since they are all members of the same process group.

In addition to the basic job control operations already mentioned, a job control-cognizant shell needs to perform the following actions.

When a foreground (not background) job stops, the shell must sample and remember the current terminal settings so that it can restore them later when it continues the stopped job in the foreground (via the *tcgetattr*( ) and *tcsetattr*( ) functions).

Because a shell itself can be spawned from a shell, it must take special action to ensure that subshells interact well with their parent shells.

A subshell can be spawned to perform an interactive function (prompting the terminal for commands) or a non-interactive function (reading commands from a file). When operating non-interactively, the job control shell will refrain from performing the job control-specific actions described above. It will behave as a shell that does not support job control. For example, all *jobs* will be left in the same process group as the shell, which itself remains in the process group established for it by its parent. This allows the shell and its children to be treated as a single job by a parent shell, and they can be affected as a unit by terminal keyboard signals.

An interactive subshell can be spawned from another job control-cognizant shell in either the foreground or background. (For example, from the C Shell, the user can execute the command, csh &.) Before the subshell activates job control by calling *setpgid*( ) to place itself in its own process group and *tcsetpgrp*( ) to place its new process group in the foreground, it needs to ensure that it has already been placed in the foreground by its parent. (Otherwise, there could be multiple job control shells that simultaneously attempt to control mediation of the terminal.) To determine this, the shell retrieves its own process group via *getpgrp*( ) and the process group of the current foreground job via *tcgetpgrp*( ). If these are not equal, the shell sends SIGTTIN to its own process group, causing itself to stop. When continued later by its parent, the shell repeats the process group check. When the process groups finally match, the shell is in the foreground and it can proceed to take control. After this point, the shell ignores all the job control stop signals so that it does not inadvertently stop itself.

Implementing Job Control Applications

Most applications do not need to be aware of job control signals and operations; the intuitively correct behavior happens by default. However, sometimes an application can inadvertently interfere with normal job control processing, or an application may choose to overtly effect job control in cooperation with normal shell procedures.

An application can inadvertently subvert job control processing by ‘‘blindly’’ altering the handling of signals. A common application error is to learn how many signals the system supports and to ignore or catch them all. Such an application makes the assumption that it does not know what this signal is, but knows the right handling action for it. The system may initialize the handling of job control stop signals so that they are being ignored. This allows shells that do not support job control to inherit and propagate these settings and hence to be immune to stop signals. A job control shell will set the handling to the default action and propagate this, allowing processes to stop. In doing so, the job control shell is taking responsibility for restarting the stopped applications. If an application wishes to catch the stop signals itself, it should first determine their inherited handling states. If a stop signal is being ignored, the application should continue to ignore it. This is directly analogous to the recommended handling of SIGINT described in the referenced UNIX Programmer’s Manual.

If an application is reading the terminal and has disabled the interpretation of special characters (by clearing the ISIG flag), the terminal I/O driver will not send SIGTSTP when the suspend character is typed. Such an application can simulate the effect of the suspend character by recognizing it and sending SIGTSTP to its process group as the terminal driver would have done. Note that the signal is sent to the process group, not just to the application itself; this ensures that other processes in the job also stop. (Note also that other processes in the job could be children, siblings, or even ancestors.) Applications should not assume that the suspend character is <control>-Z (or any particular value); they should retrieve the current setting at startup.

Implementing Job Control Systems

The intent in adding 4.2 BSD-style job control functionality was to adopt the necessary 4.2 BSD programmatic interface with only minimal changes to resolve syntactic or semantic conflicts with System V or to close recognized security holes. The goal was to maximize the ease of providing both conforming implementations and Conforming POSIX.1 Applications.

It is only useful for a process to be affected by job control signals if it is the descendant of a job control shell. Otherwise, there will be nothing that continues the stopped process.

POSIX.1 does not specify how controlling terminal access is affected by a user logging out (that is, by a controlling process terminating). 4.2 BSD uses the *vhangup*( ) function to prevent any access to the controlling terminal through file descriptors opened prior to logout. System V does not prevent controlling terminal access through file descriptors opened prior to logout (except for the case of the special file, **/dev/tty**). Some implementations choose to make processes immune from job control after logout (that is, such processes are always treated as if in the foreground); other implementations continue to enforce foreground/background checks after logout. Therefore, a Conforming POSIX.1 Application should not attempt to access the controlling terminal after logout since such access is unreliable. If an implementation chooses to deny access to a controlling terminal after its controlling process exits, POSIX.1 requires a certain type of behavior (see **Controlling Terminal** (on page 3305)).

Kernel*

See *system call*.

Library Routine*

See *system call*.

Logical Device*

Implementation-defined.

Map

The definition of map is included to clarify the usage of mapped pages in the description of the behavior of process memory locking.

Memory-Resident

The term *memory-resident* is historically understood to mean that the so-called resident pages are actually present in the physical memory of the computer system and are immune from swapping, paging, copy-on-write faults, and so on. This is the actual intent of IEEE Std 1003.1-200x in the process memory locking section for implementations where this is logical. But for some implementations—primarily mainframes—actually locking pages into primary storage is not advantageous to other system objectives, such as maximizing throughput. For such implementations, memory locking is a ‘‘hint’’ to the implementation that the application wishes to avoid situations that would cause long latencies in accessing memory. Furthermore, there are other implementation-defined issues with minimizing memory access latencies that ‘‘memory residency’’ does not address—such as MMU reload faults. The definition attempts to accommodate various implementations while allowing conforming applications to specify to the implementation that they want or need the best memory access times that the implementation can provide.

Memory Object*

The term *memory object* usually implies shared memory. If the object is the same as a filename in the file system name space of the implementation, it is expected that the data written into the memory object be preserved on disk. A memory object may also apply to a physical device on an implementation. In this case, writes to the memory object are sent to the controller for the device and reads result in control registers being returned.

Mount Point*

The directory on which a *mounted file system* is mounted. This term, like *mount*( ) and *umount*( ), was not included because it was implementation-defined.

Mounted File System*

See *file system*.

Name

There are no explicit limits in IEEE Std 1003.1-200x on the sizes of names, words (see the definition of word in the Base Definitions volume of IEEE Std 1003.1-200x), lines, or other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE_MAX} and the sum of exported variables comes under the {ARG_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a character at a time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical practice, such as makefiles, where *make* removes the <newline>s associated with the commands for a target and presents the shell with one very long line. The text on INPUT FILES in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 1.11, Utility Description Defaults does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

Native Implementation*

This refers to an implementation of POSIX.1 that interfaces directly to an operating system kernel; see also *hosted implementation* and *cooperating implementation*. A similar concept is a native UNIX system, which would be a kernel derived from one of the original UNIX system products.

Nice Value

This definition is not intended to suggest that all processes in a system have priorities that are comparable. Scheduling policy extensions, such as adding realtime priorities, make the notion of a single underlying priority for all scheduling policies problematic. Some implementations may implement the features related to *nice* to affect all processes on the system, others to affect just the general time-sharing activities implied by IEEE Std 1003.1-200x, and others may have no effect at all. Because of the use of ‘‘implementation-defined’’ in *nice* and *renice*, a wide range of implementation strategies is possible.

Open File Description

An *open file description*, as it is currently named, describes how a file is being accessed. What is currently called a *file descriptor* is actually just an identifier or ‘‘handle’’; it does not actually describe anything.

The following alternate names were discussed:

- For *open file description*:
  open instance, *file access description*, *open file information*, and *file access information*.
- For *file descriptor*:
  file handle, *file number* (c.f., *fileno*( )). Some historical implementations use the term *file table entry*.

Orphaned Process Group

Historical implementations have a concept of an orphaned process, which is a process whose parent process has exited. When job control is in use, it is necessary to prevent processes from being stopped in response to interactions with the terminal after they no longer are controlled by a job control-cognizant program. Because signals generated by the terminal are sent to a process group and not to individual processes, and because a signal may be provoked by a process that is not orphaned, but sent to another process that is orphaned, it is necessary to define an orphaned process group. The definition assumes that a process group will be manipulated as a group and that the job control-cognizant process controlling the group is outside of the group and is the parent of at least one process in the group (so that state changes may be reported via *waitpid*( )). Therefore, a group is considered to be controlled as long as at least one process in the group has a parent that is outside of the process group, but within the session.

This definition of orphaned process groups ensures that a session leader’s process group is always considered to be orphaned, and thus it is prevented from stopping in response to terminal signals.

Page

The term *page* is defined to support the description of the behavior of memory mapping for shared memory and memory mapped files, and the description of the behavior of process memory locking. It is not intended to imply that shared memory/file mapping and memory locking are applicable only to ‘‘paged’’ architectures. For the purposes of IEEE Std 1003.1-200x, whatever the granularity on which an architecture supports mapping or locking is considered to be a ‘‘page’’ . If an architecture cannot support the memory mapping or locking functions specified by IEEE Std 1003.1-200x on any granularity, then these options will not be implemented on the architecture.

Passwd File*

Implementation-defined; see **User Database** (on page 3322).

Parent Directory

There may be more than one directory entry pointing to a given directory in some implementations. The wording here identifies that exactly one of those is the parent directory. In *pathname resolution*, dot-dot is identified as the way that the unique directory is identified. (That is, the parent directory is the one to which dot-dot points.) In the case of a remote file system, if the same file system is mounted several times, it would appear as if they were distinct file systems (with interesting synchronization properties).

Pipe

It proved convenient to define a pipe as a special case of a FIFO, even though historically the latter was not introduced until System III and does not exist at all in 4.3 BSD.

Portable Filename Character Set

The encoding of this character set is not specified—specifically, ASCII is not required. But the implementation must provide a unique character code for each of the printable graphics specified by POSIX.1; see also Section A.4.6 (on page 3324).

Situations where characters beyond the portable filename character set (or historically ASCII or the ISO/IEC 646:1991 standard) would be used (in a context where the portable filename character set or the ISO/IEC 646:1991 standard is required by POSIX.1) are expected to be common. Although such a situation renders the use technically non-compliant, mutual agreement among the users of an extended character set will make such use portable between those users. Such a mutual agreement could be formalized as an optional extension to POSIX.1. (Making it required would eliminate too many possible systems, as even those systems using the ISO/IEC 646:1991 standard as a base character set extend their character sets for Western Europe and the rest of the world in different ways.)

Nothing in POSIX.1 is intended to preclude the use of extended characters where interchange is not required or where mutual agreement is obtained. It has been suggested that in several places ‘‘should’’ be used instead of ‘‘shall’’. Because (in the worst case) use of any character beyond the portable filename character set would render the program or data not portable to all possible systems, no extensions are permitted in this context.

Regular File

POSIX.1 does not intend to preclude the addition of structuring data (for example, record lengths) in the file, as long as such data is not visible to an application that uses the features described in POSIX.1.

Root Directory

This definition permits the operation of *chroot*( ), even though that function is not in POSIX.1; see also *file hierarchy*.

Root File System*

Implementation-defined.

Root of a File System*

Implementation-defined; see *mount point*.

Signal

The definition implies a double meaning for the term. Although a signal is an event, common usage implies that a signal is an identifier of the class of event.

Superuser*

This concept, with great historical significance to UNIX system users, has been replaced with the notion of appropriate privileges.

Supplementary Group ID

The POSIX.1-1990 standard is inconsistent in its treatment of supplementary groups. The definition of supplementary group ID explicitly permits the effective group ID to be included in the set, but wording in the description of the *setuid*( ) and *setgid*( ) functions states: ‘‘Any supplementary group IDs of the calling process remain unchanged by these function calls’’. In the case of *setgid*( ) this contradicts that definition. In addition, some felt that the unspecified behavior in the definition of supplementary group IDs adds unnecessary portability problems. The standard developers considered several solutions to this problem:

1. Reword the description of *setgid*( ) to permit it to change the supplementary group IDs to reflect the new effective group ID. A problem with this is that it adds more ‘‘may’’s to the wording and does not address the portability problems of this optional behavior.
2. Mandate the inclusion of the effective group ID in the supplementary set (giving {NGROUPS_MAX} a minimum value of 1). This is the behavior of 4.4 BSD. In that system, the effective group ID is the first element of the array of supplementary group IDs (there is no separate copy stored, and changes to the effective group ID are made only in the supplementary group set). By convention, the initial value of the effective group ID is duplicated elsewhere in the array so that the initial value is not lost when executing a set-group-ID program.
3. Change the definition of supplementary group ID to exclude the effective group ID and specify that the effective group ID does not change the set of supplementary group IDs. This is the behavior of 4.2 BSD, 4.3 BSD, and System V, Release 4.
4. Change the definition of supplementary group ID to exclude the effective group ID, and require that *getgroups*( ) return the union of the effective group ID and the supplementary group IDs.
5. Change the definition of {NGROUPS_MAX} to be one more than the number of supplementary group IDs, so it continues to be the number of values returned by *getgroups*( ) and existing applications continue to work. This alternative is effectively the same as the second (and might actually have the same implementation).

The standard developers decided to permit either 2 or 3. The effective group ID is orthogonal to the set of supplementary group IDs, and it is implementation-defined whether *getgroups*( ) returns this. If the effective group ID is returned with the set of supplementary group IDs, then all changes to the effective group ID affect the supplementary group set returned by *getgroups*( ). It is permissible to eliminate duplicates from the list returned by *getgroups*( ). However, if a group ID is contained in the set of supplementary group IDs, setting the group ID to that value and then to a different value should not remove that value from the supplementary group IDs.

The definition of supplementary group IDs has been changed to not include the effective group ID. This simplifies permanent rationale and makes the relevant functions easier to understand. The *getgroups*( ) function has been modified so that it can, on an implementation-defined basis, return the effective group ID. By making this change, functions that modify the effective group ID do not need to discuss adding to the supplementary group list; the only view into the supplementary group list that the application writer has is through the *getgroups*( ) function.

Symbolic Link

Many implementations associate no attributes, including ownership with symbolic links. Security experts encouraged consideration for defining these attributes as optional. Consideration was given to changing *utime*( ) to allow modification of the times for a symbolic link, or as an alternative adding an *lutime*( ) interface. Modifications to *chown*( ) were also considered: allow changing symbolic link ownership or alternatively adding *lchown*( ). As a result of the problems encountered in defining attributes for symbolic links (and interfaces to access/modify those attributes) and since implementations exist that do not associate these attributes with symbolic links, only the file type bits in the *st_mode* member and the *st_size* member of the **stat** structure are required to be applicable to symbolic links.

Historical implementations were followed when determining which interfaces should apply to symbolic links. Interfaces that historically followed symbolic links include *chmod*( ), *link*( ), and *utime*( ). Interfaces that historically do not follow symbolic links include *chown*( ), *lstat*( ), *readlink*( ), *rename*( ), *remove*( ), *rmdir*( ), and *unlink*( ). IEEE Std 1003.1-200x deviates from historical practice only in the case of *chown*( ). Because there is no requirement that there be an association of ownership with symbolic links, there was no point in requiring an interface to change ownership. In addition, other implementations of symbolic links have modified *chown*( ) to follow symbolic links.

In the case of symbolic links, IEEE Std 1003.1-200x states that a trailing slash is considered to be the final component of a pathname rather than the pathname component that preceded it. This is the behavior of historical implementations. For example, for **/a/b** and **/a/b/**, if **/a/b** is a symbolic link to a directory, then **/a/b** refers to the symbolic link, and **/a/b/** is the same as **/a/b/.**, which is the directory to which the symbolic link points.

For multi-level security purposes, it is possible to have the link read mode govern permission for the *readlink*( ) function. It is also possible that the read permissions of the directory containing the link be used for this purpose. Implementations may choose to use either of these methods; however, this is not current practice and neither method is specified.

Several reasons were advanced for requiring that when a symbolic link is used as the source argument to the *link*( ) function, the resulting link will apply to the file named by the contents of the symbolic link rather than to the symbolic link itself. This is the case in historical implementations. This action was preferred, as it supported the traditional idea of persistence with respect to the target of a hard link. This decision is appropriate in light of a previous decision not to require association of attributes with symbolic links, thereby allowing implementations which do not use inodes. Opposition centered on the lack of symmetry on the part of the *link*( ) and *unlink*( ) function pair with respect to symbolic links.

Because a symbolic link and its referenced object coexist in the file system name space, confusion can arise in distinguishing between the link itself and the referenced object. Historically, utilities and system calls have adopted their own link following conventions in a somewhat *ad hoc* fashion. Rules for a uniform approach are outlined here, although historical practice has been adhered to as much as was possible. To promote consistent system use, user-written utilities are encouraged to follow these same rules.

Symbolic links are handled either by operating on the link itself, or by operating on the object referenced by the link. In the latter case, an application or system call is said to follow the link. Symbolic links may reference other symbolic links, in which case links are dereferenced until an object that is not a symbolic link is found, a symbolic link that references a file that does not exist is found, or a loop is detected. (Current implementations do not detect loops, but have a limit on the number of symbolic links that they will dereference before declaring it an error.)

There are four domains for which default symbolic link policy is established in a system. In almost all cases, there are utility options that override this default behavior. The four domains are as follows:

1. Symbolic links specified to system calls that take filename arguments
2. Symbolic links specified as command line filename arguments to utilities that are not performing a traversal of a file hierarchy
3. Symbolic links referencing files not of type directory, specified to utilities that are performing a traversal of a file hierarchy
4. Symbolic links referencing files of type directory, specified to utilities that are performing a traversal of a file hierarchy

First Domain

The first domain is considered in earlier rationale.

Second Domain

The reason this category is restricted to utilities that are not traversing the file hierarchy is that some standard utilities take an option that specifies a hierarchical traversal, but by default operate on the arguments themselves. Generally, users specifying the option for a file hierarchy traversal wish to operate on a single, physical hierarchy, and therefore symbolic links, which may reference files outside of the hierarchy, are ignored. For example, *chown owner file* is a different operation from the same command with the –**R** option specified. In this example, the behavior of the command *chown owner file* is described here, while the behavior of the command *chown* –**R** *owner file* is described in the third and fourth domains.

The general rule is that the utilities in this category follow symbolic links named as arguments.

Exceptions in the second domain are:

- The *mv* and *rm* utilities do not follow symbolic links named as arguments, but respectively attempt to rename or delete them.
- The *ls* utility is also an exception to this rule. For compatibility with historical systems, when the –**R** option is not specified, the *ls* utility follows symbolic links named as arguments if the –**L** option is specified or if the –**F**, –**d**, or –**l** options are not specified. (If the –**L** option is specified, *ls* always follows symbolic links; it is the only utility where the –**L** option affects its behavior even though a tree walk is not being performed.)

All other standard utilities, when not traversing a file hierarchy, always follow symbolic links named as arguments.

Historical practice is that the –**h** option is specified if standard utilities are to act upon symbolic links instead of upon their targets. Examples of commands that have historically had a –**h** option for this purpose are the *chgrp*, *chown*, *file*, and *test* utilities.

Third Domain

The third domain is symbolic links, referencing files not of type directory, specified to utilities that are performing a traversal of a file hierarchy. (This includes symbolic links specified as command line filename arguments or encountered during the traversal.)

The intention of the Shell and Utilities volume of IEEE Std 1003.1-200x is that the operation that the utility is performing is applied to the symbolic link itself, if that operation is applicable to symbolic links. The reason that the operation is not required is that symbolic links in some implementations do not have such attributes as a file owner, and therefore the *chown* operation would be meaningless. If symbolic links on the system have an owner, it is the intention that the utility *chown* cause the owner of the symbolic link to change. If symbolic links do not have an owner, the symbolic link should be ignored. Specifically, by default, no change should be made to the file referenced by the symbolic link.

Fourth Domain

The fourth domain is symbolic links referencing files of type directory, specified to utilities that are performing a traversal of a file hierarchy. (This includes symbolic links specified as command line filename arguments or encountered during the traversal.)

Most standard utilities do not, by default, indirect into the file hierarchy referenced by the symbolic link. (The Shell and Utilities volume of IEEE Std 1003.1-200x uses the informal term *physical walk* to describe this case. The case where the utility does indirect through the symbolic link is termed a *logical walk*.)

There are three reasons for the default to a physical walk:

1. With very few exceptions, a physical walk has been the historical default on UNIX systems supporting symbolic links. Because some utilities (that is, *rm*) must default to a physical walk, regardless, changing historical practice in this regard would be confusing to users and needlessly incompatible.

2. For systems where symbolic links have the historical file attributes (that is, *owner*, *group*, *mode*), defaulting to a logical traversal would require the addition of a new option to the commands to modify the attributes of the link itself. This is painful and more complex than the alternatives.

3. There is a security issue with defaulting to a logical walk. Historically, the command *chown* **–R** *user file* has been safe for the superuser because *setuid* and *setgid* bits were lost when the ownership of the file was changed. If the walk were logical, changing ownership would no longer be safe because a user might have inserted a symbolic link pointing to any file in the tree. Again, this would necessitate the addition of an option to the commands doing hierarchy traversal to not indirect through the symbolic links, and historical scripts doing recursive walks would instantly become security problems. While this is mostly an issue for system administrators, it is preferable to not have different defaults for different classes of users.

As consistently as possible, users may cause standard utilities performing a file hierarchy traversal to follow any symbolic links named on the command line, regardless of the type of file they reference, by specifying the **–H** (for half logical) option. This option is intended to make the command line name space look like the logical name space.

As consistently as possible, users may cause standard utilities performing a file hierarchy traversal to follow any symbolic links named on the command line as well as any symbolic links encountered during the traversal, regardless of the type of file they reference, by specifying the **–L** (for logical) option. This option is intended to make the entire name space look like the logical name space.

For consistency, implementors are encouraged to use the −**P** (for physical) flag to specify the physical walk in utilities that do logical walks by default for whatever reason. The only standard utilities that require the −**P** option are *cd* and *pwd*; see the note below.

When one or more of the −**H**, −**L**, and −**P** flags can be specified, the last one specified determines the behavior of the utility. This permits users to alias commands so that the default behavior is a logical walk and then override that behavior on the command line.

Exceptions in the Third and Fourth Domains

The *ls* and *rm* utilities are exceptions to these rules. The *rm* utility never follows symbolic links and does not support the −**H**, −**L**, or −**P** options. Some historical versions of *ls* always followed symbolic links given on the command line whether the −**L** option was specified or not. Historical versions of *ls* did not support the −**H** option. In IEEE Std 1003.1-200x, unless one of the −**H** or −**L** options is specified, the *ls* utility only follows symbolic links to directories that are given as operands. The *ls* utility does not support the −**P** option.

The Shell and Utilities volume of IEEE Std 1003.1-200x requires that the standard utilities *ls*, *find*, and *pax* detect infinite loops when doing logical walks; that is, a directory, or more commonly a symbolic link, that refers to an ancestor in the current file hierarchy. If the file system itself is corrupted, causing the infinite loop, it may be impossible to recover. Because *find* and *ls* are often used in system administration and security applications, they should attempt to recover and continue as best as they can. The *pax* utility should terminate because the archive it was creating is by definition corrupted. Other, less vital, utilities should probably simply terminate as well. Implementations are strongly encouraged to detect infinite loops in all utilities.

Historical practice is shown in Table A-1 (on page 3320). The heading **SVID3** stands for the Third Edition of the System V Interface Definition.

Historically, several shells have had built-in versions of the *pwd* utility. In some of these shells, *pwd* reported the physical path, and in others, the logical path. Implementations of the shell corresponding to IEEE Std 1003.1-200x must report the logical path by default. Earlier versions of IEEE Std 1003.1-200x did not require the *pwd* utility to be a built-in utility. Now that *pwd* is required to set an environment variable in the current shell execution environment, it must be a built-in utility.

The *cd* command is required, by default, to treat the filename dot-dot logically. Implementors are required to support the −**P** flag in *cd* so that users can have their current environment handled physically. In 4.3 BSD, *chgrp* during tree traversal changed the group of the symbolic link, not the target. Symbolic links in 4.4 BSD do not have *owner*, *group*, *mode*, or other standard UNIX system file attributes.

Table A-1 Historical Practice for Symbolic Links

Utility	SVID3	4.3 BSD	4.4 BSD	POSIX	Comments
cd				**–L**	Treat `".."` logically.
cd				**–P**	`".."` physically.
chgrp			**–H**	**–H**	Follow command line symlinks.
chgrp			**–h**	**–L**	Follow symlinks.
chgrp	**–h**			**–h**	Affect the symlink.
chmod					Affect the symlink.
chmod			**–H**		Follow command line symlinks.
chmod			**–h**		Follow symlinks.
chown			**–H**	**–H**	Follow command line symlinks.
chown			**–h**	**–L**	Follow symlinks.
chown	**–h**			**–h**	Affect the symlink.
cp			**–H**	**–H**	Follow command line symlinks.
cp			**–h**	**–L**	Follow symlinks.
cpio	**–L**		**–L**		Follow symlinks.
du			**–H**	**–H**	Follow command line symlinks.
du			**–h**	**–L**	Follow symlinks.
file	**–h**			**–h**	Affect the symlink.
find			**–H**	**–H**	Follow command line symlinks.
find			**–h**	**–L**	Follow symlinks.
find	**–follow**		**–follow**		Follow symlinks.
ln	**–s**	**–s**	**–s**	**–s**	Create a symbolic link.
ls	**–L**	**–L**	**–L**	**–L**	Follow symlinks.
ls				**–H**	Follow command line symlinks.
mv					Operates on the symlink.
pax			**–H**	**–H**	Follow command line symlinks.
pax			**–h**	**–L**	Follow symlinks.
pwd				**–L**	Printed path may contain symlinks.
pwd				**–P**	Printed path will not contain symlinks.
rm					Operates on the symlink.
tar			**–H**		Follow command line symlinks.
tar		**–h**	**–h**		Follow symlinks.
test	**–h**		**–h**	**–h**	Affect the symlink.

Synchronously-Generated Signal

Those signals that may be generated synchronously include SIGABRT, SIGBUS, SIGILL, SIGFPE, SIGPIPE, and SIGSEGV.

Any signal sent via the *raise*( ) function or a *kill*( ) function targeting the current process is also considered synchronous.

System Call*

The distinction between a *system call* and a *library routine* is an implementation detail that may differ between implementations and has thus been excluded from POSIX.1.

See ‘‘Interface, Not Implementation’’ in the Preface.

System Reboot

A *system reboot* is an event initiated by an unspecified circumstance that causes all processes (other than special system processes) to be terminated in an implementation-defined manner, after which any changes to the state and contents of files created or written to by a Conforming POSIX.1 Application prior to the event are implementation-defined.

Synchronized I/O Data (and File) Integrity Completion

These terms specify that for synchronized read operations, pending writes must be successfully completed before the read operation can complete. This is motivated by two circumstances. Firstly, when synchronizing processes can access the same file, but not share common buffers (such as for a remote file system), this requirement permits the reading process to guarantee that it can read data written remotely. Secondly, having data written synchronously is insufficient to guarantee the order with respect to a subsequent write by a reading process, and thus this extra read semantic is necessary.

Text File

The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters or they explicitly describe their limitations within their individual descriptions. The definition of *text file* has caused controversy. The only difference between text and binary files is that text files have lines of less than {LINE_MAX} bytes, with no NUL characters, each terminated by a <newline>. The definition allows a file with a single <newline>, but not a totally empty file, to be called a text file. If a file ends with an incomplete line it is not strictly a text file by this definition. The <newline> referred to in IEEE Std 1003.1-200x is not some generic line separator, but a single character; files created on systems where they use multiple characters for ends of lines are not portable to all conforming systems without some translation process unspecified by IEEE Std 1003.1-200x.

Thread

IEEE Std 1003.1-200x defines a thread to be a flow of control within a process. Each thread has a minimal amount of private state; most of the state associated with a process is shared among all of the threads in the process. While most multi-thread extensions to POSIX have taken this approach, others have made different decisions.

Note: The choice to put threads within a process does not constrain implementations to implement threads in that manner. However, all functions have to behave as though threads share the indicated state information with the process from which they were created.

Threads need to share resources in order to cooperate. Memory has to be widely shared between threads in order for the threads to cooperate at a fine level of granularity. Threads keep data structures and the locks protecting those data structures in shared memory. For a data structure to be usefully shared between threads, such structures should not refer to any data that can only be interpreted meaningfully by a single thread. Thus, any system resources that might be referred to in data structures need to be shared between all threads. File descriptors, pathnames, and pointers to stack variables are all things that programmers want to share between their threads. Thus, the file descriptor table, the root directory, the current working directory, and the address space have to be shared.

Library implementations are possible as long as the effective behavior is as if system services invoked by one thread do not suspend other threads. This may be difficult for some library implementations on systems that do not provide asynchronous facilities.

See Section B.2.9 (on page 3439) for additional rationale.

Thread ID

See Section B.2.9.2 (on page 3455) for additional rationale.

Thread-Safe Function

All functions required by IEEE Std 1003.1-200x need to be thread-safe; see Section A.4.16 (on page 3330) and Section B.2.9.1 (on page 3452) for additional rationale.

User Database

There are no references in IEEE Std 1003.1-200x to a *passwd file* or a *group file*, and there is no requirement that the *group* or *passwd* databases be kept in files containing editable text. Many large timesharing systems use *passwd* databases that are hashed for speed. Certain security classifications prohibit certain information in the *passwd* database from being publicly readable.

The term *encoded* is used instead of *encrypted* in order to avoid the implementation connotations (such as reversibility or use of a particular algorithm) of the latter term.

The *getgrent*( ), *setgrent*( ), *endgrent*( ), *getpwent*( ), *setpwent*( ), and *endpwent*( ) functions are not included as part of the base standard because they provide a linear database search capability that is not generally useful (the *getpwuid*( ), *getpwnam*( ), *getgrgid*( ), and *getgrnam*( ) functions are provided for keyed lookup) and because in certain distributed systems, especially those with different authentication domains, it may not be possible or desirable to provide an application with the ability to browse the system databases indiscriminately. They are provided on XSI-conformant systems due to their historical usage by many existing applications.

A change from historical implementations is that the structures used by these functions have fields of the types **gid_t** and **uid_t**, which are required to be defined in the **<sys/types.h>** header. IEEE Std 1003.1-200x requires implementations to ensure that these types are defined by inclusion of **<grp.h>** and **<pwd.h>**, respectively, without imposing any name space pollution or errors from redefinition of types.

IEEE Std 1003.1-200x is silent about the content of the strings containing user or group names. These could be digit strings. IEEE Std 1003.1-200x is also silent as to whether such digit strings bear any relationship to the corresponding (numeric) user or group ID.

Database Access

The thread-safe versions of the user and group database access functions return values in user-supplied buffers instead of possibly using static data areas that may be overwritten by each call.

Virtual Processor*

The term *virtual processor* was chosen as a neutral term describing all kernel-level schedulable entities, such as processes, Mach tasks, or lightweight processes. Implementing threads using multiple processes as virtual processors, or implementing multiplexed threads above a virtual processor layer, should be possible, provided some mechanism has also been implemented for sharing state between processes or virtual processors. Many systems may also wish to provide implementations of threads on systems providing ‘‘shared processes’’ or ‘‘variable-weight processes’’. It was felt that exposing such implementation details would severely limit the type of systems upon which the threads interface could be supported and prevent certain types of valid implementations. It was also determined that a virtual processor interface was out of the scope of the Rationale (Informative) volume of IEEE Std 1003.1-200x.

XSI

This is introduced to allow IEEE Std 1003.1-200x to be adopted as an IEEE standard and an Open Group Technical Standard, serving both the POSIX and the Single UNIX Specification in a core set of volumes.

The term *XSI* has been used for 10 years in connection with the XPG series and the first and second versions of the base volumes of the Single UNIX Specification. The XSI margin code was introduced to denote the extended or more restrictive semantics beyond POSIX that are applicable to UNIX systems.

A.4 General Concepts

A.4.1 Concurrent Execution

There is no additional rationale provided for this section.

A.4.2 Directory Protection

There is no additional rationale provided for this section.

A.4.3 Extended Security Controls

Allowing an implementation to define extended security controls enables the use of IEEE Std 1003.1-200x in environments that require different or more rigorous security than that provided in POSIX.1. Extensions are allowed in two areas: privilege and file access permissions. The semantics of these areas have been defined to permit extensions with reasonable, but not exact, compatibility with all existing practices. For example, the elimination of the superuser definition precludes identifying a process as privileged or not by virtue of its effective user ID.

A.4.4 File Access Permissions

A process should not try to anticipate the result of an attempt to access data by *a priori* use of these rules. Rather, it should make the attempt to access data and examine the return value (and possibly *errno* as well), or use *access*( ). An implementation may include other security mechanisms in addition to those specified in POSIX.1, and an access attempt may fail because of those additional mechanisms, even though it would succeed according to the rules given in this section. (For example, the user's security level might be lower than that of the object of the access attempt.) The supplementary group IDs provide another reason for a process to not attempt to anticipate the result of an access attempt.

A.4.5 File Hierarchy

Though the file hierarchy is commonly regarded to be a tree, POSIX.1 does not define it as such for three reasons:

1. Links may join branches.
2. In some network implementations, there may be no single absolute root directory; see *pathname resolution*.
3. With symbolic links, the file system need not be a tree or even a directed acyclic graph.

A.4.6 Filenames

Historically, certain filenames have been reserved. This list includes **core**, **/etc/passwd**, and so on. Conforming applications should avoid these.

Most historical implementations prohibit case folding in filenames; that is, treating uppercase and lowercase alphabetic characters as identical. However, some consider case folding desirable:

- For user convenience
- For ease-of-implementation of the POSIX.1 interface as a hosted system on some popular operating systems

Variants, such as maintaining case distinctions in filenames, but ignoring them in comparisons, have been suggested. Methods of allowing escaped characters of the case opposite the default have been proposed.

Many reasons have been expressed for not allowing case folding, including:

- No solid evidence has been produced as to whether case-sensitivity or case-insensitivity is more convenient for users.
- Making case-insensitivity a POSIX.1 implementation option would be worse than either having it or not having it, because:
  - — More confusion would be caused among users.
  - — Application developers would have to account for both cases in their code.
  - — POSIX.1 implementors would still have other problems with native file systems, such as short or otherwise constrained filenames or pathnames, and the lack of hierarchical directory structure.
- Case folding is not easily defined in many European languages, both because many of them use characters outside the US ASCII alphabetic set, and because:
  - — In Spanish, the digraph `"ll"` is considered to be a single letter, the capitalized form of which may be either `"Ll"` or `"LL"`, depending on context.
  - — In French, the capitalized form of a letter with an accent may or may not retain the accent, depending on the country in which it is written.
  - — In German, the sharp ess may be represented as a single character resembling a Greek beta (β) in lowercase, but as the digraph `"SS"` in uppercase.
  - — In Greek, there are several lowercase forms of some letters; the one to use depends on its position in the word. Arabic has similar rules.
- Many East Asian languages, including Japanese, Chinese, and Korean, do not distinguish case and are sometimes encoded in character sets that use more than one byte per character.
- Multiple character codes may be used on the same machine simultaneously. There are several ISO character sets for European alphabets. In Japan, several Japanese character codes are commonly used together, sometimes even in filenames; this is evidently also the case in China. To handle case insensitivity, the kernel would have to at least be able to distinguish for which character sets the concept made sense.
- The file system implementation historically deals only with bytes, not with characters, except for slash and the null byte.
- The purpose of POSIX.1 is to standardize the common, existing definition, not to change it. Mandating case-insensitivity would make all historical implementations non-standard.

- Not only the interface, but also application programs would need to change, counter to the purpose of having minimal changes to existing application code.
- At least one of the original developers of the UNIX system has expressed objection in the strongest terms to either requiring case-insensitivity or making it an option, mostly on the basis that POSIX.1 should not hinder portability of application programs across related implementations in order to allow compatibility with unrelated operating systems.

Two proposals were entertained regarding case folding in filenames:

1. Remove all wording that previously permitted case folding.

   Rationale Case folding is inconsistent with portable filename character set definition and filename definition (all characters except slash and null). No known implementations allowing all characters except slash and null also do case folding.

2. Change ‘‘though this practice is not recommended:’’ to ‘‘although this practice is strongly discouraged.’’

   Rationale If case folding must be included in POSIX.1, the wording should be stronger to discourage the practice.

The consensus selected the first proposal. Otherwise, a conforming application would have to assume that case folding would occur when it was not wanted, but that it would not occur when it was wanted.

A.4.7 File Times Update

This section reflects the actions of historical implementations. The times are not updated immediately, but are only marked for update by the functions. An implementation may update these times immediately.

The accuracy of the time update values is intentionally left unspecified so that systems can control the bandwidth of a possible covert channel.

The wording was carefully chosen to make it clear that there is no requirement that the conformance document contain information that might incidentally affect file update times. Any function that performs pathname resolution might update several *st_atime* fields. Functions such as *getpwnam*( ) and *getgrnam*( ) might update the *st_atime* field of some specific file or files. It is intended that these are not required to be documented in the conformance document, but they should appear in the system documentation.

A.4.8 Host and Network Byte Order

There is no additional rationale provided for this section.

A.4.9 Measurement of Execution Time

The methods used to measure the execution time of processes and threads, and the precision of these measurements, may vary considerably depending on the software architecture of the implementation, and on the underlying hardware. Implementations can also make tradeoffs between the scheduling overhead and the precision of the execution time measurements. IEEE Std 1003.1-200x does not impose any requirement on the accuracy of the execution time; it instead specifies that the measurement mechanism and its precision are implementation-defined.

A.4.10 Memory Synchronization

In older multi-processors, access to memory by the processors was strictly multiplexed. This meant that a processor executing program code interrogates or modifies memory in the order specified by the code and that all the memory operation of all the processors in the system appear to happen in some global order, though the operation histories of different processors are interleaved arbitrarily. The memory operations of such machines are said to be sequentially consistent. In this environment, threads can synchronize using ordinary memory operations. For example, a producer thread and a consumer thread can synchronize access to a circular data buffer as follows:

```
int rdptr = 0;
int wrptr = 0;
data_t buf[BUFSIZE];

Thread 1:
    while (work_to_do) {
        int next;

        buf[wrptr] = produce();
        next = (wrptr + 1) % BUFSIZE;
        while (rdptr == next)
            ;
        wrptr = next;
}

Thread 2:
    while (work_to_do) {
        while (rdptr == wrptr)
            ;
        consume(buf[rdptr]);
        rdptr = (rdptr + 1) % BUFSIZE;
    }
```

In modern multi-processors, these conditions are relaxed to achieve greater performance. If one processor stores values in location A and then location B, then other processors loading data from location B and then location A may see the new value of B but the old value of A. The memory operations of such machines are said to be weakly ordered. On these machines, the circular buffer technique shown in the example will fail because the consumer may see the new value of *wrptr* but the old value of the data in the buffer. In such machines, synchronization can only be achieved through the use of special instructions that enforce an order on memory operations. Most high-level language compilers only generate ordinary memory operations to take advantage of the increased performance. They usually cannot determine when memory operation order is important and generate the special ordering instructions. Instead, they rely on the programmer to use synchronization primitives correctly to ensure that modifications to a location in memory are ordered with respect to modifications and/or access to the same location in other threads. Access to read-only data need not be synchronized. The resulting program is said to be data race-free.

Synchronization is still important even when accessing a single primitive variable (for example, an integer). On machines where the integer may not be aligned to the bus data width or be larger than the data width, a single memory load may require multiple memory cycles. This means that it may be possible for some parts of the integer to have an old value while other parts have a newer value. On some processor architectures this cannot happen, but portable programs cannot rely on this.

In summary, a portable multi-threaded program, or a multi-process program that shares writable memory between processes, has to use the synchronization primitives to synchronize data access. It cannot rely on modifications to memory being observed by other threads in the order written in the program or even on modification of a single variable being seen atomically.

Conforming applications may only use the functions listed to synchronize threads of control with respect to memory access. There are many other candidates for functions that might also be used. Examples are: signal sending and reception, or pipe writing and reading. In general, any function that allows one thread of control to wait for an action caused by another thread of control is a candidate. IEEE Std 1003.1-200x does not require these additional functions to synchronize memory access since this would imply the following:

- All these functions would have to be recognized by advanced compilation systems so that memory operations and calls to these functions are not reordered by optimization.
- All these functions would potentially have to have memory synchronization instructions added, depending on the particular machine.
- The additional functions complicate the model of how memory is synchronized and make automatic data race detection techniques impractical.

Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective. IEEE Std 1003.1-200x defines functions that can be used to synchronize access to memory, but it leaves open exactly how one relates those functions to the semantics of each function as specified elsewhere in IEEE Std 1003.1-200x. IEEE Std 1003.1-200x also does not make a formal specification of the partial ordering in time that the functions can impose, as that is implied in the description of the semantics of each function. It simply states that the programmer has to ensure that modifications do not occur ‘‘simultaneously’’ with other access to a memory location.

A.4.11 Pathname Resolution

It is necessary to differentiate between the definition of pathname and the concept of pathname resolution with respect to the handling of trailing slashes. By specifying the behavior here, it is not possible to provide an implementation that is conforming but extends all interfaces that handle pathnames to also handle strings that are not legal pathnames (because they have trailing slashes).

Pathnames that end with one or more trailing slash characters must refer to directory paths. Previous versions of IEEE Std 1003.1-200x were not specific about the distinction between trailing slashes on files and directories, and both were permitted.

Two types of implementation have been prevalent; those that ignored trailing slash characters on all pathnames regardless, and those that only permitted them only on existing directories.

IEEE Std 1003.1-200x requires that a pathname with a trailing slash character be treated as if it had a trailing `"/."` everywhere.

Note that this change does not break any conforming applications; since there were two different types of implementation, no application could have portably depended on either behavior. This change does however require some implementations to be altered to remain compliant. Substantial discussion over a three-year period has shown that the benefits to application developers outweighs the disadvantages for some vendors. |

On a historical note, some early applications automatically appended a `'/'` to every path. Rather than fix the applications, the system implementation was modified to accept this behavior by ignoring any trailing slash.

Each directory has exactly one parent directory which is represented by the name **dot-dot** in the first directory. No other directory, regardless of linkages established by symbolic links, is considered the parent directory by IEEE Std 1003.1-200x.

There are two general categories of interfaces involving pathname resolution: those that follow the symbolic link, and those that do not. There are several exceptions to this rule; for example, *open*(*path*,O_CREAT | O_EXCL) will fail when *path* names a symbolic link. However, in all other situations, the *open*( ) function will follow the link.

What the filename **dot-dot** refers to relative to the root directory is implementation-defined. In Version 7 it refers to the root directory itself; this is the behavior mentioned in IEEE Std 1003.1-200x. In some networked systems the construction **/../hostname/** is used to refer to the root directory of another host, and POSIX.1 permits this behavior.

Other networked systems use the construct **//hostname** for the same purpose; that is, a double initial slash is used. There is a potential problem with existing applications that create full pathnames by taking a trunk and a relative pathname and making them into a single string separated by `'/'`, because they can accidentally create networked pathnames when the trunk is `'/'`. This practice is not prohibited because such applications can be made to conform by simply changing to use `"//"` as a separator instead of `'/'`:

- If the trunk is `'/'`, the full pathname will begin with `"///"` (the initial `'/'` and the separator `"//"`). This is the same as `'/'`, which is what is desired. (This is the general case of making a relative pathname into an absolute one by prefixing with `"///"` instead of `'/'`.)
- If the trunk is `"/A"`, the result is `"/A//..."`; since non-leading sequences of two or more slashes are treated as a single slash, this is equivalent to the desired `"/A/..."`.
- If the trunk is `"//A"`, the implementation-defined semantics will apply. (The multiple slash rule would apply.)

Application developers should avoid generating pathnames that start with `"//"`. Implementations are strongly encouraged to avoid using this special interpretation since a number of applications currently do not follow this practice and may inadvertently generate `"//..."`.

The term *root directory* is only defined in POSIX.1 relative to the process. In some implementations, there may be no absolute root directory. The initialization of the root directory of a process is implementation-defined.

A.4.12 Process ID Reuse

There is no additional rationale provided for this section.

A.4.13 Scheduling Policy

There is no additional rationale provided for this section.

A.4.14 Seconds Since the Epoch

Coordinated Universal Time (UTC) includes leap seconds. However, in POSIX time (seconds since the Epoch), leap seconds are ignored (not applied) to provide an easy and compatible method of computing time differences. Broken-down POSIX time is therefore not necessarily UTC, despite its appearance.

As of September 2000, 24 leap seconds had been added to UTC since the Epoch, 1 January, 1970. Historically, one leap second is added every 15 months on average, so this offset can be expected to grow steadily with time.

Most systems' notion of ''time'' is that of a continuously increasing value, so this value should increase even during leap seconds. However, not only do most systems not keep track of leap seconds, but most systems are probably not synchronized to any standard time reference. Therefore, it is inappropriate to require that a time represented as seconds since the Epoch precisely represent the number of seconds between the referenced time and the Epoch.

It is sufficient to require that applications be allowed to treat this time as if it represented the number of seconds between the referenced time and the Epoch. It is the responsibility of the vendor of the system, and the administrator of the system, to ensure that this value represents the number of seconds between the referenced time and the Epoch as closely as necessary for the application being run on that system.

It is important that the interpretation of time names and *seconds since the Epoch* values be consistent across conforming systems; that is, it is important that all conforming systems interpret ''536 457 599 seconds since the Epoch'' as 59 seconds, 59 minutes, 23 hours 31 December 1986, regardless of the accuracy of the system's idea of the current time. The expression is given to assure a consistent interpretation, not to attempt to specify the calendar. The relationship between *tm_yday* and the day of week, day of month, and month is presumed to be specified elsewhere and is not given in POSIX.1.

Consistent interpretation of *seconds since the Epoch* can be critical to certain types of distributed applications that rely on such timestamps to synchronize events. The accrual of leap seconds in a time standard is not predictable. The number of leap seconds since the Epoch will likely increase. POSIX.1 is more concerned about the synchronization of time between applications of astronomically short duration.

Note that *tm_yday* is zero-based, not one-based, so the day number in the example above is 364. Note also that the division is an integer division (discarding remainder) as in the C language.

Note also that the meaning of *gmtime*( ), *localtime*( ), and *mktime*( ) is specified in terms of this expression. However, the ISO C standard computes *tm_yday* from *tm_mday*, *tm_mon*, and *tm_year* in *mktime*( ). Because it is stated as a (bidirectional) relationship, not a function, and because the conversion between month-day-year and day-of-year dates is presumed well known and is also a relationship, this is not a problem.

Implementations that implement **time_t** as a signed 32-bit integer will overflow in 2 038. The data size for **time_t** is as per the ISO C standard definition, which is implementation-defined.

See also **Epoch** (on page 3306).

The topic of whether seconds since the Epoch should account for leap seconds has been debated on a number of occasions, and each time consensus was reached (with acknowledged dissent each time) that the majority of users are best served by treating all days identically. (That is, the majority of applications were judged to assume a single length—as measured in seconds since the Epoch—for all days. Thus, leap seconds are not applied to seconds since the Epoch.) Those applications which do care about leap seconds can determine how to handle them in whatever way those applications feel is best. This was particularly emphasized because there was disagreement about what the best way of handling leap seconds might be. It is a practical impossibility to mandate that a conforming implementation must have a fixed relationship to any particular official clock (consider isolated systems, or systems performing ‘‘reruns’’ by setting the clock to some arbitrary time).

Note that as a practical consequence of this, the length of a second as measured by some external standard is not specified. This unspecified second is nominally equal to an International System (SI) second in duration. Applications must be matched to a system that provides the particular handling of external time in the way required by the application.

A.4.15 Semaphore

There is no additional rationale provided for this section.

A.4.16 Thread-Safety

Where the interface of a function required by IEEE Std 1003.1-200x precludes thread-safety, an alternate form that shall be thread-safe is provided. The names of these thread-safe forms are the same as the non-thread-safe forms with the addition of the suffix ‘‘_r’’. The suffix ‘‘_r’’ is historical, where the `'r'` stood for ‘‘reentrant’’.

In some cases, thread-safety is provided by restricting the arguments to an existing function.

See also Section B.2.9.1 (on page 3452).

A.4.17 Tracing

Refer to Section B.2.11 (on page 3468).

A.4.18 Treatment of Error Conditions for Mathematical Functions

There is no additional rationale provided for this section.

A.4.19 Treatment of NaN Arguments for Mathematical Functions

There is no additional rationale provided for this section.

A.4.20 Utility

There is no additional rationale provided for this section.

A.4.21 Variable Assignment

There is no additional rationale provided for this section.

A.5 File Format Notation

The notation for spaces allows some flexibility for application output. Note that an empty character position in *format* represents one or more <blank>s on the output (not *white space*, which can include <newline>s). Therefore, another utility that reads that output as its input must be prepared to parse the data using *scanf*( ), *awk*, and so on. The `'Δ'` character is used when exactly one <space> is output.

The treatment of integers and spaces is different from the *printf*( ) function in that they can be surrounded with <blank>s. This was done so that, given a format such as:

```
"%d\n",<foo>
```

the implementation could use a *printf*( ) call such as:

```
printf("%6d\n", foo);
```

and still conform. This notation is thus somewhat like *scanf*( ) in addition to *printf*( ).

The *printf*( ) function was chosen as a model because most of the standard developers were familiar with it. One difference from the C function *printf*( ) is that the `l` and `h` conversion specifier characters are not used. As expressed by the Shell and Utilities volume of IEEE Std 1003.1-200x, there is no differentiation between decimal values for type **int**, type **long**, or type **short**. The conversion specifications `%d` or `%i` should be interpreted as an arbitrary length sequence of digits. Also, no distinction is made between single precision and double precision numbers (**float** or **double** in C). These are simply referred to as floating-point numbers.

Many of the output descriptions in the Shell and Utilities volume of IEEE Std 1003.1-200x use the term *line*, such as:

```
"%s", <input line>
```

Since the definition of *line* includes the trailing <newline> already, there is no need to include a `'\n'` in the format; a double <newline> would otherwise result.

A.6 Character Set

A.6.1 Portable Character Set

The portable character set is listed in full so there is no dependency on the ISO/IEC 646:1991 standard (or historically ASCII) encoded character set, although the set is identical to the characters defined in the International Reference version of the ISO/IEC 646:1991 standard.

IEEE Std 1003.1-200x poses no requirement that multiple character sets or codesets be supported, leaving this as a marketing differentiation for implementors. Although multiple charmap files are supported, it is the responsibility of the implementation to provide the file(s); if only one is provided, only that one will be accessible using the *localedef* **−f** option.

The statement about invariance in codesets for the portable character set is worded to avoid precluding implementations where multiple incompatible codesets are available (for instance, ASCII and EBCDIC). The standard utilities cannot be expected to produce predictable results if they access portable characters that vary on the same implementation.

Not all character sets need include the portable character set, but each locale must include it. For example, a Japanese-based locale might be supported by a mixture of character sets: JIS X 0201 Roman (a Japanese version of the ISO/IEC 646:1991 standard), JIS X 0208, and JIS X 0201 Katakana. Not all of these character sets include the portable characters, but at least one does (JIS X 0201 Roman).

A.6.2 Character Encoding

Encoding mechanisms based on single shifts, such as the EUC encoding used in some Asian and other countries, can be supported via the current charmap mechanism. With single-shift encoding, each character is preceded by a shift code (SS2 or SS3). A complete EUC code, consisting of the portable character set (G0) and up to three additional character sets (G1, G2, G3), can be described using the current charmap mechanism; the encoding for each character in additional character sets G2 and G3 must then include their single-shift code. Other mechanisms to support locales based on encoding mechanisms such as locking shift are not addressed by this volume of IEEE Std 1003.1-200x.

A.6.3 C Language Wide-Character Codes

There is no additional rationale provided for this section.

A.6.4 Character Set Description File

IEEE PASC Interpretation 1003.2 #196 is applied, removing three lines of text dealing with ranges of symbolic names using position constant values which had been erroneously included in the final 1003.2b draft. |

A.6.4.1 State-Dependent Character Encodings

A requirement was considered that would force utilities to eliminate any redundant locking shifts, but this was left as a quality of implementation issue.

This change satisfies the following requirement from the ISO POSIX-2: 1993 standard, Annex H.1:

> *The support of state-dependent (shift encoding) character sets should be addressed fully. See descriptions of these in the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.2, Character Encoding. If such character encodings are supported, it is expected that this will impact the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.2, Character Encoding, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 7, Locale, the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions , and the comm, cut, diff, grep, head, join, paste, and tail utilities.*

The character set description file provides:

- The capability to describe character set attributes (such as collation order or character classes) independent of character set encoding, and using only the characters in the portable character set. This makes it possible to create generic *localedef* source files for all codesets that share the portable character set (such as the ISO 8859 family or IBM Extended ASCII).
- Standardized symbolic names for all characters in the portable character set, making it possible to refer to any such character regardless of encoding.

Implementations are free to choose their own symbolic names, as long as the names identified by this volume of IEEE Std 1003.1-200x are also defined; this provides support for already existing ‘‘character names’’.

The names selected for the members of the portable character set follow the ISO/IEC 8859-1: 1998 standard and the ISO/IEC 10646-1: 2000 standard. However, several commonly used UNIX system names occur as synonyms in the list:

- The historical UNIX system names are used for control characters.

- The word ‘‘slash’’ is given in addition to ‘‘solidus’’.
- The word ‘‘backslash’’ is given in addition to ‘‘reverse-solidus’’.
- The word ‘‘hyphen’’ is given in addition to ‘‘hyphen-minus’’.
- The word ‘‘period’’ is given in addition to ‘‘full-stop’’.
- For digits, the word ‘‘digit’’ is eliminated.
- For letters, the words ‘‘Latin Capital Letter’’ and ‘‘Latin Small Letter’’ are eliminated.
- The words ‘‘left brace’’ and ‘‘right brace’’ are given in addition to ‘‘left-curly-bracket’’ and ‘‘right-curly-bracket’’.
- The names of the digits are preferred over the numbers to avoid possible confusion between `'0'` and `'O'`, and between `'1'` and `'l'` (one and the letter ell).

The names for the control characters in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 6, Character Set were taken from the ISO/IEC 4873: 1991 standard.

The charmap file was introduced to resolve problems with the portability of, especially, *localedef* sources. IEEE Std 1003.1-200x assumes that the portable character set is constant across all locales, but does not prohibit implementations from supporting two incompatible codings, such as both ASCII and EBCDIC. Such dual-support implementations should have all charmaps and *localedef* sources encoded using one portable character set, in effect cross-compiling for the other environment. Naturally, charmaps (and *localedef* sources) are only portable without transformation between systems using the same encodings for the portable character set. They can, however, be transformed between two sets using only a subset of the actual characters (the portable character set). However, the particular coded character set used for an application or an implementation does not necessarily imply different characteristics or collation; on the contrary, these attributes should in many cases be identical, regardless of codeset. The charmap provides the capability to define a common locale definition for multiple codesets (the same *localedef* source can be used for codesets with different extended characters; the ability in the charmap to define empty names allows for characters missing in certain codesets).

The **<escape_char>** declaration was added at the request of the international community to ease the creation of portable charmap files on terminals not implementing the default backslash escape. The **<comment_char>** declaration was added at the request of the international community to eliminate the potential confusion between the number sign and the pound sign.

The octal number notation with no leading zero required was selected to match those of *awk* and *tr* and is consistent with that used by *localedef*. To avoid confusion between an octal constant and the back-references used in *localedef* source, the octal, hexadecimal, and decimal constants shall contain at least two digits. As single-digit constants are relatively rare, this should not impose any significant hardship. Provision is made for more digits to account for systems in which the byte size is larger than 8 bits. For example, a Unicode (ISO/IEC 10646-1: 2000 standard) system that has defined 16-bit bytes may require six octal, four hexadecimal, and five decimal digits.

The decimal notation is supported because some newer international standards define character values in decimal, rather than in the old column/row notation.

The charmap identifies the coded character sets supported by an implementation. At least one charmap shall be provided, but no implementation is required to provide more than one. Likewise, implementations can allow users to generate new charmaps (for instance, for a new version of the ISO 8859 family of coded character sets), but does not have to do so. If users are allowed to create new charmaps, the system documentation describes the rules that apply (for instance, ‘‘only coded character sets that are supersets of the ISO/IEC 646: 1991 standard IRV, no multi-byte characters'').

This addition of the **WIDTH** specification satisfies the following requirement from the ISO POSIX-2: 1993 standard, Annex H.1:

> (9) *The definition of column position relies on the implementation's knowledge of the integral width of the characters. The charmap or LC_CTYPE locale definitions should be enhanced to allow application specification of these widths.*

The character ''width'' information was first considered for inclusion under *LC_CTYPE* but was moved because it is more closely associated with the information in the *charmap* than information in the locale source (cultural conventions information). Concerns were raised that formalizing this type of information is moving the locale source definition from the codeset-independent entity that it was designed to be to a repository of codeset-specific information. A similar issue occurred with the **<code_set_name>**, **<mb_cur_max>**, and **<mb_cur_min>** information, which was resolved to reside in the *charmap* definition.

The width definition was added to the IEEE P1003.2b draft standard with the intent that the *wcswidth*( ) and/or *wcwidth*( ) functions (currently specified in the System Interfaces volume of IEEE Std 1003.1-200x) be the mechanism to retrieve the character width information.

A.7 Locale

A.7.1 General

The description of locales is based on work performed in the UniForum Technical Committee Subcommittee on Internationalization. Wherever appropriate, keywords are taken from the ISO C standard or the X/Open Portability Guide.

The value used to specify a locale with environment variables is the name specified as the *name* operand to the *localedef* utility when the locale was created. This provides a verifiable method to create and invoke a locale.

The ''object'' definitions need not be portable, as long as ''source'' definitions are. Strictly speaking, source definitions are portable only between implementations using the same character set(s). Such source definitions, if they use symbolic names only, easily can be ported between systems using different codesets, as long as the characters in the portable character set (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set) have common values between the codesets; this is frequently the case in historical implementations. Of source, this requires that the symbolic names used for characters outside the portable character set be identical between character sets. The definition of symbolic names for characters is outside the scope of IEEE Std 1003.1-200x, but is certainly within the scope of other standards organizations.

Applications can select the desired locale by invoking the *setlocale*( ) function (or equivalent) with the appropriate value. If the function is invoked with an empty string, the value of the corresponding environment variable is used. If the environment variable is not set or is set to the empty string, the implementation sets the appropriate environment as defined in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 8, Environment Variables.

A.7.2 POSIX Locale

The POSIX locale is equal to the C locale. To avoid being classified as a C-language function, the name has been changed to the POSIX locale; the environment variable value can be either `"POSIX"` or, for historical reasons, `"C"`.

The POSIX definitions mirror the historical UNIX system behavior.

The use of symbolic names for characters in the tables does not imply that the POSIX locale must be described using symbolic character names, but merely that it may be advantageous to do so.

A.7.3 Locale Definition

The decision to separate the file format from the *localedef* utility description was only partially editorial. Implementations may provide other interfaces than *localedef*. Requirements on ‘‘the utility’’, mostly concerning error messages, are described in this way because they are meant to affect the other interfaces implementations may provide as well as *localedef*.

The text about POSIX2_LOCALEDEF does not mean that internationalization is optional; only that the functionality of the *localedef* utility is. REs, for instance, must still be able to recognize, for example, character class expressions such as `"[[:alpha:]]"`. A possible analogy is with an applications development environment; while all conforming implementations must be capable of executing applications, not all need to have the development environment installed. The assumption is that the capability to modify the behavior of utilities (and applications) via locale settings must be supported. If the *localedef* utility is not present, then the only choice is to select an existing (presumably implementation-documented) locale. An implementation could, for example, choose to support only the POSIX locale, which would in effect limit the amount of changes from historical implementations quite drastically. The *localedef* utility is still required, but would always terminate with an exit code indicating that no locale could be created. Supported locales must be documented using the syntax defined in this chapter. (This ensures that users can accurately determine what capabilities are provided. If the implementation decides to provide additional capabilities to the ones in this chapter, that is already provided for.)

If the option is present (that is, locales can be created), then the *localedef* utility must be capable of creating locales based on the syntax and rules defined in this chapter. This does not mean that the implementation cannot also provide alternate means for creating locales.

The octal, decimal, and hexadecimal notations are the same employed by the charmap facility (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.4, Character Set Description File). To avoid confusion between an octal constant and a back-reference, the octal, hexadecimal, and decimal constants must contain at least two digits. As single-digit constants are relatively rare, this should not impose any significant hardship. Provision is made for more digits to account for systems in which the byte size is larger than 8 bits. For example, a Unicode (see the ISO/IEC 10646-1:2000 standard) system that has defined 16-bit bytes may require six octal, four hexadecimal, and five decimal digits. As with the charmap file, multi-byte characters are described in the locale definition file using ‘‘big-endian’’ notation for reasons of portability. There is no requirement that the internal representation in the computer memory be in this same order.

One of the guidelines used for the development of this volume of IEEE Std 1003.1-200x is that characters outside the invariant part of the ISO/IEC 646:1991 standard should not be used in portable specifications. The backslash character is not in the invariant part; the number sign is, but with multiple representations: as a number sign, and as a pound sign. As far as general usage of these symbols, they are covered by the ‘‘grandfather clause’’, but for newly defined interfaces, the WG15 POSIX working group has requested that POSIX provide alternate representations. Consequently, while the default escape character remains the backslash and the default comment character is the number sign, implementations are required to recognize alternative representations, identified in the applicable source file via the **<escape_char>** and **<comment_char>** keywords.

A.7.3.1 LC_CTYPE

The *LC_CTYPE* category is primarily used to define the encoding-independent aspects of a character set, such as character classification. In addition, certain encoding-dependent characteristics are also defined for an application via the *LC_CTYPE* category. IEEE Std 1003.1-200x does not mandate that the encoding used in the locale is the same as the one used by the application because an implementation may decide that it is advantageous to define locales in a system-wide encoding rather than having multiple, logically identical locales in different encodings, and to convert from the application encoding to the system-wide encoding on usage. Other implementations could require encoding-dependent locales.

In either case, the *LC_CTYPE* attributes that are directly dependent on the encoding, such as **<mb_cur_max>** and the display width of characters, are not user-specifiable in a locale source and are consequently not defined as keywords.

Implementations may define additional keywords or extend the *LC_CTYPE* mechanism to allow application-defined keywords.

The text ‘‘The ellipsis specification shall only be valid within a single encoded character set’’ is present because it is possible to have a locale supported by multiple character encodings, as explained in the rationale for the Base Definitions volume of IEEE Std 1003.1-200x, Section 6.1, Portable Character Set. An example given there is of a possible Japanese-based locale supported by a mixture of the character sets JIS X 0201 Roman, JIS X 0208, and JIS X 0201 Katakana. Attempting to express a range of characters across these sets is not logical and the implementation is free to reject such attempts.

As the *LC_CTYPE* character classes are based on the ISO C standard character class definition, the category does not support multi-character elements. For instance, the German character <sharp-s> is traditionally classified as a lowercase letter. There is no corresponding uppercase letter; in proper capitalization of German text, the <sharp-s> will be replaced by "SS"; that is, by two characters. This kind of conversion is outside the scope of the **toupper** and **tolower** keywords.

Where IEEE Std 1003.1-200x specifies that only certain characters can be specified, as for the keywords **digit** and **xdigit**, the specified characters shall be from the portable character set, as shown. As an example, only the Arabic digits 0 through 9 are acceptable as digits.

The character classes **digit**, **xdigit**, **lower**, **upper**, and **space** have a set of automatically included characters. These only need to be specified if the character values (that is, encoding) differs from the implementation default values. It is not possible to define a locale without these automatically included characters unless some implementation extension is used to prevent their inclusion. Such a definition would not be a proper superset of the C locale, and thus, it might not be possible for the standard utilities to be implemented as programs conforming to the ISO C standard.

The definition of character class **digit** requires that only ten characters—the ones defining digits—can be specified; alternate digits (for example, Hindi or Kanji) cannot be specified here. However, the encoding may vary if an implementation supports more than one encoding.

The definition of character class **xdigit** requires that the characters included in character class **digit** are included here also and allows for different symbols for the hexadecimal digits 10 through 15.

The inclusion of the **charclass** keyword satisfies the following requirement from the ISO POSIX-2: 1993 standard, Annex H.1:

(3) *The LC_CTYPE (2.5.2.1) locale definition should be enhanced to allow user-specified additional character classes, similar in concept to the ISO C standard Multibyte Support Extension (MSE) is_wctype( ) function.*

This keyword was previously included in The Open Group specifications and is now mandated in the Shell and Utilities volume of IEEE Std 1003.1-200x.

The symbolic constant {CHARCLASS_NAME_MAX} was also adopted from The Open Group specifications. Application portability is enhanced by the use of symbolic constants.

A.7.3.2 LC_COLLATE

The rules governing collation depend to some extent on the use. At least five different levels of increasingly complex collation rules can be distinguished:

1. *Byte/machine code order*: This is the historical collation order in the UNIX system and many proprietary operating systems. Collation is here performed character by character, without any regard to context. The primary virtue is that it usually is quite fast and also completely deterministic; it works well when the native machine collation sequence matches the user expectations.

2. *Character order*: On this level, collation is also performed character by character, without regard to context. The order between characters is, however, not determined by the code values, but on the expectations by the user of the ‘‘correct’’ order between characters. In addition, such a (simple) collation order can specify that certain characters collate equally (for example, uppercase and lowercase letters).

3. *String ordering*: On this level, entire strings are compared based on relatively straightforward rules. Several ‘‘passes’’ may be required to determine the order between two strings. Characters may be ignored in some passes, but not in others; the strings may be compared in different directions; and simple string substitutions may be performed before strings are compared. This level is best described as ‘‘dictionary’’ ordering; it is based on the spelling, not the pronunciation, or meaning, of the words.

4. *Text search ordering*: This is a further refinement of the previous level, best described as ‘‘telephone book ordering’’; some common homonyms (words spelled differently but with the same pronunciation) are collated together; numbers are collated as if they were spelled out, and so on.

5. *Semantic-level ordering*: Words and strings are collated based on their meaning; entire words (such as ‘‘the’’) are eliminated; the ordering is not deterministic. This usually requires special software and is highly dependent on the intended use.

While the historical collation order formally is at level 1, for the English language it corresponds roughly to elements at level 2. The user expects to see the output from the *ls* utility sorted very much as it would be in a dictionary. While telephone book ordering would be an optimal goal for standard collation, this was ruled out as the order would be language-dependent. Furthermore, a requirement was that the order must be determined solely from the text string and the collation rules; no external information (for example, ‘‘pronunciation dictionaries’’) could be required.

As a result, the goal for the collation support is at level 3. This also matches the requirements for the Canadian collation order, as well as other, known collation requirements for alphabetic scripts. It specifically rules out collation based on pronunciation rules or based on semantic analysis of the text.

The syntax for the *LC_COLLATE* category source meets the requirements for level 3 and has been verified to produce the correct result with examples based on French, Canadian, and Danish collation order. Because it supports multi-character collating elements, it is also capable of supporting collation in codesets where a character is expressed using non-spacing characters followed by the base character (such as the ISO / IEC 6937: 1994 standard).

The directives that can be specified in an operand to the **order_start** keyword are based on the requirements specified in several proposed standards and in customary use. The following is a rephrasing of rules defined for ‘‘lexical ordering in English and French’’ by the Canadian Standards Association (the text in square brackets is rephrased):

- Once special characters [punctuation] have been removed from original strings, the ordering is determined by scanning forwards (left to right) [disregarding case and diacriticals].
- In case of equivalence, special characters are once again removed from original strings and the ordering is determined by scanning backwards (starting from the rightmost character of the string and back), character by character [disregarding case but considering diacriticals].
- In case of repeated equivalence, special characters are removed again from original strings and the ordering is determined by scanning forwards, character by character [considering both case and diacriticals].
- If there is still an ordering equivalence after the first three rules have been applied, then only special characters and the position they occupy in the string are considered to determine ordering. The string that has a special character in the lowest position comes first. If two strings have a special character in the same position, the character [with the lowest collation value] comes first. In case of equality, the other special characters are considered until there is a difference or until all special characters have been exhausted.

It is estimated that this part of IEEE Std 1003.1-200x covers the requirements for all European languages, and no particular problems are anticipated with Slavic or Middle East character sets.

The Far East (particularly Japanese / Chinese) collations are often based on contextual information and pronunciation rules (the same ideogram can have different meanings and different pronunciations). Such collation, in general, falls outside the desired goal of IEEE Std 1003.1-200x. There are, however, several other collation rules (stroke / radical or ‘‘most common pronunciation’’) that can be supported with the mechanism described here.

The character order is defined by the order in which characters and elements are specified between the **order_start** and **order_end** keywords. Weights assigned to the characters and elements define the collation sequence; in the absence of weights, the character order is also the collation sequence.

The **position** keyword provides the capability to consider, in a compare, the relative position of characters not subject to **IGNORE**. As an example, consider the two strings `"o-ring"` and `"or-ing"`. Assuming the hyphen is subject to **IGNORE** on the first pass, the two strings compare equal, and the position of the hyphen is immaterial. On second pass, all characters except the hyphen are subject to **IGNORE**, and in the normal case the two strings would again compare equal. By taking position into account, the first collates before the second.

A.7.3.3 LC_MONETARY

The currency symbol does not appear in *LC_MONETARY* because it is not defined in the C locale of the ISO C standard.

The ISO C standard limits the size of decimal points and thousands delimiters to single-byte values. In locales based on multi-byte coded character sets, this cannot be enforced; IEEE Std 1003.1-200x does not prohibit such characters, but makes the behavior unspecified (in the text ''In contexts where other standards ...'').

The grouping specification is based on, but not identical to, the ISO C standard. The −1 signals that no further grouping shall be performed; the equivalent of {CHAR_MAX} in the ISO C standard.

The text ''the value is not available in the locale'' is taken from the ISO C standard and is used instead of the ''unspecified'' text in early proposals. There is no implication that omitting these keywords or assigning them values of `" "` or −1 produces unspecified results; such omissions or assignments eliminate the effects described for the keyword or produce zero-length strings, as appropriate.

The locale definition is an extension of the ISO C standard *localeconv*( ) specification. In particular, rules on how **currency_symbol** is treated are extended to also cover **int_curr_symbol**, and **p_set_by_space** and **n_sep_by_space** have been augmented with the value 2, which places a <space> between the sign and the symbol (if they are adjacent; otherwise, it should be treated as a 0). The following table shows the result of various combinations:

		p_sep_by_space		
		2	**1**	**0**
p_cs_precedes = 1	**p_sign_posn** = 0	`($1.25)`	`($ 1.25)`	`($1.25)`
	p_sign_posn = 1	`+ $1.25`	`+$ 1.25`	`+$1.25`
	p_sign_posn = 2	`$1.25 +`	`$ 1.25+`	`$1.25+`
	p_sign_posn = 3	`+ $1.25`	`+$ 1.25`	`+$1.25`
	p_sign_posn = 4	`$ +1.25`	`$+ 1.25`	`$+1.25`
p_cs_precedes = 0	**p_sign_posn** = 0	`(1.25 $)`	`(1.25 $)`	`(1.25$)`
	p_sign_posn = 1	`+1.25 $`	`+1.25 $`	`+1.25$`
	p_sign_posn = 2	`1.25$ +`	`1.25 $+`	`1.25$+`
	p_sign_posn = 3	`1.25+ $`	`1.25 +$`	`1.25+$`
	p_sign_posn = 4	`1.25$ +`	`1.25 $+`	`1.25$+`

The following is an example of the interpretation of the **mon_grouping** keyword. Assuming that the value to be formatted is 123456789 and the **mon_thousands_sep** is `'''`, then the following table shows the result. The third column shows the equivalent string in the ISO C standard that would be used by the *localeconv*( ) function to accommodate this grouping.

mon_grouping	**Formatted Value**	**ISO C String**
3;−1	123456'789	`"\3\177"`
3	123'456'789	`"\3"`
3;2;−1	1234'56'789	`"\3\2\177"`
3;2	12'34'56'789	`"\3\2"`
−1	123456789	`"\177"`

In these examples, the octal value of {CHAR_MAX} is 177.

A.7.3.4 *LC_NUMERIC*

See the rationale for *LC_MONETARY* for a description of the behavior of grouping.

A.7.3.5 *LC_TIME*

Although certain of the conversion specifications in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these conversion specifications may need to adjust the capitalization if the output is going to be used at the beginning of a sentence.

The *LC_TIME* descriptions of **abday**, **day**, **mon**, and **abmon** imply a Gregorian style calendar (7-day weeks, 12-month years, leap years, and so on). Formatting time strings for other types of calendars is outside the scope of IEEE Std 1003.1-200x.

While the ISO 8601: 2000 standard numbers the weekdays starting with Monday, historical practice is to use the Sunday as the first day. Rather than change the order and introduce potential confusion, the days must be specified beginning with Sunday; previous references to ‘‘first day’’ have been removed. Note also that the Shell and Utilities volume of IEEE Std 1003.1-200x *date* utility supports numbering compliant with the ISO 8601: 2000 standard. |

As specified under *date* in the Shell and Utilities volume of IEEE Std 1003.1-200x and *strftime*( ) in the System Interfaces volume of IEEE Std 1003.1-200x, the conversion specifications corresponding to the optional keywords consist of a modifier followed by a traditional conversion specification (for instance, `%Ex`). If the optional keywords are not supported by the implementation or are unspecified for the current locale, these modified conversion specifications are treated as the traditional conversion specifications. For example, assume the following keywords:

```
alt_digits   "0th";"1st";"2nd";"3rd";"4th";"5th";\
             "6th";"7th";"8th";"9th";"10th"

d_fmt        "The %Od day of %B in %Y"
```

On July 4th 1776, the `%x` conversion specifications would result in `"The 4th day of July in 1776"`, while on July 14th 1789 it would result in `"The 14 day of July in 1789"`. It can be noted that the above example is for illustrative purposes only; the `%O` modifier is primarily intended to provide for Kanji or Hindi digits in *date* formats.

The following is an example for Japan that supports the current plus last three Emperors and reverts to Western style numbering for years prior to the Meiji era. The example also allows for the custom of using a special name for the first year of an era instead of using 1. (The examples substitute romaji where kanji should be used.)

```
era_d_fmt "%EY%mgatsu%dnichi (%a)"

era    "+:2:1990/01/01:+*:Heisei:%EC%Eynen";\
       "+:1:1989/01/08:1989/12/31:Heisei:%ECgannen";\
       "+:2:1927/01/01:1989/01/07:Shouwa:%EC%Eynen";\
       "+:1:1926/12/25:1926/12/31:Shouwa:%ECgannen";\
       "+:2:1913/01/01:1926/12/24:Taishou:%EC%Eynen";\
       "+:1:1912/07/30:1912/12/31:Taishou:%ECgannen";\
       "+:2:1869/01/01:1912/07/29:Meiji:%EC%Eynen";\
       "+:1:1868/09/08:1868/12/31:Meiji:%ECgannen";\
       "-:1868:1868/09/07:-*::%Ey"
```

Assuming that the current date is September 21, 1991, a request to *date* or *strftime*( ) would yield the following results:

```
%Ec - Heisei3nen9gatsu21nichi (Sat) 14:39:26
%EC - Heisei
%Ex - Heisei3nen9gatsu21nichi (Sat)
%Ey - 3
%EY - Heisei3nen
```

Example era definitions for the Republic of China:

```
era     "+:2:1913/01/01:+*:ChungHwaMingGuo:%EC%EyNen";\
        "+:1:1912/1/1:1912/12/31:ChungHwaMingGuo:%ECYuenNen";\
        "+:1:1911/12/31:-*:MingChien:%EC%EyNen"
```

Example definitions for the Christian Era:

```
era     "+:0:0001/01/01:+*:AD:%EC %Ey";\
        "+:1:-0001/12/31:-*:BC:%Ey %EC"
```

A.7.3.6 LC_MESSAGES

The **yesstr** and **nostr** locale keywords and the YESSTR and NOSTR *langinfo* items were formerly used to match user affirmative and negative responses. In IEEE Std 1003.1-200x, the **yesexpr**, **noexpr**, YESEXPR, and NOEXPR extended regular expressions have replaced them. Applications should use the general locale-based messaging facilities to issue prompting messages which include sample desired responses.

A.7.4 Locale Definition Grammar

There is no additional rationale provided for this section.

A.7.4.1 Locale Lexical Conventions

There is no additional rationale provided for this section.

A.7.4.2 Locale Grammar

There is no additional rationale provided for this section.

A.7.5 Locale Definition Example

The following is an example of a locale definition file that could be used as input to the *localedef* utility. It assumes that the utility is executed with the −**f** option, naming a *charmap* file with (at least) the following content:

```
CHARMAP
<space>      \x20
<dollar>     \x24
<A>          \101
<a>          \141
<A-acute>    \346
<a-acute>    \365
<A-grave>    \300
<a-grave>    \366
<b>          \142
<C>          \103
<c>          \143
<c-cedilla>  \347
<d>          \x64
<H>          \110
<h>          \150
<eszet>      \xb7
<s>          \x73
<z>          \x7a
END CHARMAP
```

It should not be taken as complete or to represent any actual locale, but only to illustrate the syntax.

```
#
LC_CTYPE
lower   <a>;<b>;<c>;<c-cedilla>;<d>;...;<z>
upper   A;B;C;Ç;...;Z
space   \x20;\x09;\x0a;\x0b;\x0c;\x0d
blank   \040;\011
toupper (<a>,<A>);(b,B);(c,C);(ç,Ç);(d,D);(z,Z)
END LC_CTYPE
#
LC_COLLATE
#
# The following example of collation is based on
# Canadian standard Z243.4.1-1998, "Canadian Alphanumeric
# Ordering Standard For Character sets of CSA Z234.4 Standard".
# (Other parts of this example locale definition file do not
# purport to relate to Canada, or to any other real culture.)
# The proposed standard defines a 4-weight collation, such that
# in the first pass, characters are compared without regard to
# case or accents; in second pass, backwards compare without
# regard to case; in the third pass, forward compare without
# regard to diacriticals. In the 3 first passes, non-alphabetic
# characters are ignored; in the fourth pass, only special
# characters are considered, such that "The string that has a
# special character in the lowest position comes first. If two
# strings have a special character in the same position, the
# collation value of the special character determines ordering.
#
# Only a subset of the character set is used here; mostly to
# illustrate the set-up.
```

```
#
collating-symbol <NULL>
collating-symbol <LOW_VALUE>
collating-symbol <LOWER-CASE>
collating-symbol <SUBSCRIPT-LOWER>
collating-symbol <SUPERSCRIPT-LOWER>
collating-symbol <UPPER-CASE>
collating-symbol <NO-ACCENT>
collating-symbol <PECULIAR>
collating-symbol <LIGATURE>
collating-symbol <ACUTE>
collating-symbol <GRAVE>
# Further collating-symbols follow.
#
# Properly, the standard does not include any multi-character
# collating elements; the one below is added for completeness.
#
collating_element <ch> from "<c><h>"
collating_element <CH> from "<C><H>"
collating_element <Ch> from "<C><h>"
#
order_start forward;backward;forward;forward,position
#
# Collating symbols are specified first in the sequence to allocate
# basic collation values to them, lower than that of any character.
<NULL>
<LOW_VALUE>
<LOWER-CASE>
<SUBSCRIPT-LOWER>
<SUPERSCRIPT-LOWER>
<UPPER-CASE>
<NO-ACCENT>
<PECULIAR>
<LIGATURE>
<ACUTE>
<GRAVE>
<RING-ABOVE>
<DIAERESIS>
<TILDE>
# Further collating symbols are given a basic collating value here.
#
# Here follow special characters.
<space>        IGNORE;IGNORE;IGNORE;<space>
# Other special characters follow here.
#
# Here follow the regular characters.
<a>         <a>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<A>         <a>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<a-acute>   <a>;<ACUTE>;<LOWER-CASE>;IGNORE
<A-acute>   <a>;<ACUTE>;<UPPER-CASE>;IGNORE
<a-grave>   <a>;<GRAVE>;<LOWER-CASE>;IGNORE
<A-grave>   <a>;<GRAVE>;<UPPER-CASE>;IGNORE
```

```
<ae>      "<a><e>";"<LIGATURE><LIGATURE>";\
          "<LOWER-CASE><LOWER-CASE>";IGNORE
<AE>      "<a><e>";"<LIGATURE><LIGATURE>";\
          "<UPPER-CASE><UPPER-CASE>";IGNORE
<b>        <b>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<B>        <b>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<c>        <c>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<C>        <c>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<ch>       <ch>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<Ch>       <ch>;<NO-ACCENT>;<PECULIAR>;IGNORE
<CH>       <ch>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
#
# As an example, the strings "Bach" and "bach" could be encoded (for
# compare purposes) as:
# "Bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\
#         <NO_ACCENT>;<LOW_VALUE>;<UPPER-CASE>;<LOWER-CASE>;\
#         <LOWER-CASE>;<NULL>
# "bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\
#         <NO_ACCENT>;<LOW_VALUE>;<LOWER-CASE>;<LOWER-CASE>;\
#         <LOWER-CASE>;<NULL>
#
# The two strings are equal in pass 1 and 2, but differ in pass 3.
#
# Further characters follow.
#
UNDEFINED    IGNORE;IGNORE;IGNORE;IGNORE
#
order_end
#
END LC_COLLATE
#
LC_MONETARY
int_curr_symbol    "USD "
currency_symbol    "$"
mon_decimal_point  "."
mon_grouping       3;0
positive_sign      ""
negative_sign      "-"
p_cs_precedes      1
n_sign_posn        0
END LC_MONETARY
#
LC_NUMERIC
copy "US_en.ASCII"
END LC_NUMERIC
#
LC_TIME
abday   "Sun";"Mon";"Tue";"Wed";"Thu";"Fri";"Sat"
#
day     "Sunday";"Monday";"Tuesday";"Wednesday";\
        "Thursday";"Friday";"Saturday"
#
```

```
abmon   "Jan";"Feb";"Mar";"Apr";"May";"Jun";\
         "Jul";"Aug";"Sep";"Oct";"Nov";"Dec"
#
mon     "January";"February";"March";"April";\
        "May";"June";"July";"August";"September";\
        "October";"November";"December"
#
d_t_fmt "%a %b %d %T %Z %Y\n"
END LC_TIME
#
LC_MESSAGES
yesexpr "^([yY][[:alpha:]]*)|(OK)"
#
noexpr  "^[nN][[:alpha:]]*"
END LC_MESSAGES
```

A.8 Environment Variables

A.8.1 Environment Variable Definition

The variable *environ* is not intended to be declared in any header, but rather to be declared by the user for accessing the array of strings that is the environment. This is the traditional usage of the symbol. Putting it into a header could break some programs that use the symbol for their own purposes.

The decision to restrict conforming systems to the use of digits, uppercase letters, and underscores for environment variable names allows applications to use lowercase letters in their environment variable names without conflicting with any conforming system.

In addition to the obvious conflict with the shell syntax for positional parameter substitution, some historical applications (including some shells) exclude names with leading digits from the environment.

A.8.2 Internationalization Variables

The text about locale implies that any utilities written in standard C and conforming to IEEE Std 1003.1-200x must issue the following call:

```
setlocale(LC_ALL, "")
```

If this were omitted, the ISO C standard specifies that the C locale would be used.

If any of the environment variables are invalid, it makes sense to default to an implementation-defined, consistent locale environment. It is more confusing for a user to have partial settings occur in case of a mistake. All utilities would then behave in one language/cultural environment. Furthermore, it provides a way of forcing the whole environment to be the implementation-defined default. Disastrous results could occur if a pipeline of utilities partially uses the environment variables in different ways. In this case, it would be appropriate for utilities that use *LANG* and related variables to exit with an error if any of the variables are invalid. For example, users typing individual commands at a terminal might want *date* to work if *LC_MONETARY* is invalid as long as *LC_TIME* is valid. Since these are conflicting reasonable alternatives, IEEE Std 1003.1-200x leaves the results unspecified if the locale environment variables would not produce a complete locale matching the specification of the user.

The locale settings of individual categories cannot be truly independent and still guarantee correct results. For example, when collating two strings, characters must first be extracted from each string (governed by *LC_CTYPE*) before being mapped to collating elements (governed by *LC_COLLATE*) for comparison. That is, if *LC_CTYPE* is causing parsing according to the rules of a large, multi-byte code set (potentially returning 20 000 or more distinct character codeset values), but *LC_COLLATE* is set to handle only an 8-bit codeset with 256 distinct characters, meaningful results are obviously impossible.

The *LC_MESSAGES* variable affects the language of messages generated by the standard utilities.

The description of the environment variable names starting with the characters ‘‘LC_’’ acknowledges the fact that the interfaces presented may be extended as new international functionality is required. In the ISO C standard, names preceded by ‘‘LC_’’ are reserved in the name space for future categories.

To avoid name clashes, new categories and environment variables are divided into two classifications: *implementation-independent* and *implementation-defined*.

Implementation-independent names will have the following format:

`LC_NAME`

where *NAME* is the name of the new category and environment variable. Capital letters must be used for implementation-independent names.

Implementation-defined names must be in lowercase letters, as below:

`LC_name`

A.8.3 Other Environment Variables

The quoted form of the timezone variable allows timezone names of the form UTC+1 (or any name that contains the character plus (`'+'`), the character minus (`'−'`), or digits), which may be appropriate for countries that do not have an official timezone name. It would be coded as <UTC+1>+1<UTC+2>, which would cause *std* to have a value of UTC+1 and *dst* a value of UTC+2, each with a length of 5 characters. This does not appear to conflict with any existing usage. The characters `'<'` and `'>'` were chosen for quoting because they are easier to parse visually than a quoting character that does not provide some sense of bracketing (and in a string like this, such bracketing is helpful). They were also chosen because they do not need special treatment when assigning to the *TZ* variable. Users are often confused by embedding quotes in a string. Because `'<'` and `'>'` are meaningful to the shell, the whole string would have to be quoted, but that is easily explained. (Parentheses would have presented the same problems.) Although the `'>'` symbol could have been permitted in the string by either escaping it or doubling it, it seemed of little value to require that. This could be provided as an extension if there was a need. Timezone names of this new form lead to a requirement that the value of {_POSIX_TZNAME_MAX} change from 3 to 6.

COLUMNS, LINES

The default value for the number of column positions, *COLUMNS*, and screen height, *LINES*, are unspecified because historical implementations use different methods to determine values corresponding to the size of the screen in which the utility is run. This size is typically known to the implementation through the value of *TERM*, or by more elaborate methods such as extensions to the *stty* utility or knowledge of how the user is dynamically resizing windows on a bit-mapped display terminal. Users should not need to set these variables in the environment unless there is a specific reason to override the default behavior of the implementation, such as to display data in an area arbitrarily smaller than the terminal or window. Values for these variables that are not decimal integers greater than zero are implicitly undefined values; it is unnecessary to enumerate all of the possible values outside of the acceptable set.

PATH

Many historical implementations of the Bourne shell do not interpret a trailing colon to represent the current working directory and are thus non-conforming. The C Shell and the KornShell conform to IEEE Std 1003.1-200x on this point. The usual name of dot may also be used to refer to the current working directory.

Many implementations historically have used a default value of **/bin** and **/usr/bin** for the *PATH* variable. IEEE Std 1003.1-200x does not mandate this default path be identical to that retrieved from *getconf* _CS_PATH because it is likely that the standardized utilities may be provided in another directory separate from the directories used by some historical applications.

LOGNAME

In most implementations, the value of such a variable is easily forged, so security-critical applications should rely on other means of determining user identity. *LOGNAME* is required to be constructed from the portable filename character set for reasons of interchange. No diagnostic condition is specified for violating this rule, and no requirement for enforcement exists. The intent of the requirement is that if extended characters are used, the ‘‘guarantee’’ of portability implied by a standard is void.

SHELL

The *SHELL* variable names the preferred shell of the user; it is a guide to applications. There is no direct requirement that that shell conform to IEEE Std 1003.1-200x; that decision should rest with the user. It is the intention of the standard developers that alternative shells be permitted, if the user chooses to develop or acquire one. An operating system that builds its shell into the ‘‘kernel’’ in such a manner that alternative shells would be impossible does not conform to the spirit of IEEE Std 1003.1-200x.

CHANGE HISTORY

Issue 6

Changed format of *TZ* field to allow for the quoted form as defined in previous versions of the ISO POSIX-1 standard.

A.9 Regular Expressions

Rather than repeating the description of REs for each utility supporting REs, the standard developers preferred a common, comprehensive description of regular expressions in one place. The most common behavior is described here, and exceptions or extensions to this are documented for the respective utilities, as appropriate.

The BRE corresponds to the *ed* or historical *grep* type, and the ERE corresponds to the historical *egrep* type (now *grep* −**E**).

The text is based on the *ed* description and substantially modified, primarily to aid developers and others in the understanding of the capabilities and limitations of REs. Much of this was influenced by internationalization requirements.

It should be noted that the definitions in this section do not cover the *tr* utility; the *tr* syntax does not employ REs.

The specification of REs is particularly important to internationalization because pattern matching operations are very basic operations in business and other operations. The syntax and rules of REs are intended to be as intuitive as possible to make them easy to understand and use. The historical rules and behavior do not provide that capability to non-English language users, and do not provide the necessary support for commonly used characters and language constructs. It was necessary to provide extensions to the historical RE syntax and rules to accommodate other languages.

As they are limited to bracket expressions, the rationale for these modifications is in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.5, RE Bracket Expression.

A.9.1 Regular Expression Definitions

It is possible to determine what strings correspond to subexpressions by recursively applying the leftmost longest rule to each subexpression, but only with the proviso that the overall match is leftmost longest. For example, matching `"\(ac*\)c*d[ac]*\1"` against *acdacaaa* matches *acdacaaa* (with \1=*a*); simply matching the longest match for `"\(ac*\)"` would yield \1=*ac*, but the overall match would be smaller (*acdac*). Conceptually, the implementation must examine every possible match and among those that yield the leftmost longest total matches, pick the one that does the longest match for the leftmost subexpression, and so on. Note that this means that matching by subexpressions is context-dependent: a subexpression within a larger RE may match a different string from the one it would match as an independent RE, and two instances of the same subexpression within the same larger RE may match different lengths even in similar sequences of characters. For example, in the ERE `"(a.*b)(a.*b)"`, the two identical subexpressions would match four and six characters, respectively, of *accbaccccb*.

The definition of *single character* has been expanded to include also collating elements consisting of two or more characters; this expansion is applicable only when a bracket expression is included in the BRE or ERE. An example of such a collating element may be the Dutch *ij*, which collates as a `'y'`. In some encodings, a ligature "i with j" exists as a character and would represent a single-character collating element. In another encoding, no such ligature exists, and the two-character sequence *ij* is defined as a multi-character collating element. Outside brackets, the *ij* is treated as a two-character RE and matches the same characters in a string. Historically, a bracket expression only matched a single character. The ISO POSIX-2:1993 standard required bracket expressions like `"[^[:lower:]]"` to match multi-character collating elements such as `"ij"`. However, this requirement led to behavior that many users did not expect and that could not feasibly be mimicked in user code, and it was rarely if ever implemented correctly. The current standard leaves it unspecified whether a bracket expression matches a multi-character collating element, allowing both historical and ISO POSIX-2:1993 standard implementations to conform.

Also, in the current standard, it is unspecified whether character class expressions like `"[:lower:]"` can include multi-character collating elements like `"ij"`; hence `"[[:lower:]]"` can match `"ij"`, and `"[^[:lower:]]"` can fail to match `"ij"`. Common practice is for a character class expression to match a collating element if it matches the collating element's first character.

A.9.2 Regular Expression General Requirements

The definition of which sequence is matched when several are possible is based on the leftmost-longest rule historically used by deterministic recognizers. This rule is easier to define and describe, and arguably more useful, than the first-match rule historically used by non-deterministic recognizers. It is thought that dependencies on the choice of rule are rare; carefully contrived examples are needed to demonstrate the difference.

A formal expression of the leftmost-longest rule is:

> The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

Historically, most RE implementations only match lines, not strings. However, that is more an effect of the usage than of an inherent feature of REs themselves. Consequently, IEEE Std 1003.1-200x does not regard <newline>s as special; they are ordinary characters, and both a period and a non-matching list can match them. Those utilities (like *grep*) that do not allow <newline>s to match are responsible for eliminating any <newline> from strings before matching against the RE. The *regcomp*( ) function, however, can provide support for such processing without violating the rules of this section.

Some implementations of *egrep* have had very limited flexibility in handling complex EREs. IEEE Std 1003.1-200x does not attempt to define the complexity of a BRE or ERE, but does place a lower limit on it—any RE must be handled, as long as it can be expressed in 256 bytes or less. (Of course, this does not place an upper limit on the implementation.) There are historical programs using a non-deterministic-recognizer implementation that should have no difficulty with this limit. It is possible that a good approach would be to attempt to use the faster, but more limited, deterministic recognizer for simple expressions and to fall back on the non-deterministic recognizer for those expressions requiring it. Non-deterministic implementations must be careful to observe the rules on which match is chosen; the longest match, not the first match, starting at a given character is used.

The term *invalid* highlights a difference between this section and some others: IEEE Std 1003.1-200x frequently avoids mandating of errors for syntax violations because they can be used by implementors to trigger extensions. However, the authors of the internationalization features of REs wanted to mandate errors for certain conditions to identify usage problems or non-portable constructs. These are identified within this rationale as appropriate. The remaining syntax violations have been left implicitly or explicitly undefined. For example, the BRE construct `"\{1,2,3\}"` does not comply with the grammar. A conforming application cannot rely on it producing an error nor matching the literal characters `"\{1,2,3\}"`. The term ‘‘undefined’’ was used in favor of ‘‘unspecified’’ because many of the situations are considered errors on some implementations, and the standard developers considered that consistency throughout the section was preferable to mixing undefined and unspecified.

A.9.3 Basic Regular Expressions

There is no additional rationale provided for this section.

A.9.3.1 BREs Matching a Single Character or Collating Element

There is no additional rationale provided for this section.

A.9.3.2 BRE Ordinary Characters

There is no additional rationale provided for this section.

A.9.3.3 BRE Special Characters

There is no additional rationale provided for this section.

A.9.3.4 Periods in BREs

There is no additional rationale provided for this section.

A.9.3.5 RE Bracket Expression

Range expressions are, historically, an integral part of REs. However, the requirements of ‘‘natural language behavior’’ and portability do conflict. In the POSIX locale, ranges must be treated according to the collating sequence and include such characters that fall within the range based on that collating sequence, regardless of character values. In other locales, ranges have unspecified behavior.

Some historical implementations allow range expressions where the ending range point of one range is also the starting point of the next (for instance, `"[a–m–o]"`). This behavior should not be permitted, but to avoid breaking historical implementations, it is now *undefined* whether it is a valid expression and how it should be interpreted.

Current practice in *awk* and *lex* is to accept escape sequences in bracket expressions as per the Base Definitions volume of IEEE Std 1003.1-200x, Table 5-1, Escape Sequences and Associated Actions, while the normal ERE behavior is to regard such a sequence as consisting of two characters. Allowing the *awk*/*lex* behavior in EREs would change the normal behavior in an unacceptable way; it is expected that *awk* and *lex* will decode escape sequences in EREs before passing them to *regcomp*( ) or comparable routines. Each utility describes the escape sequences it accepts as an exception to the rules in this section; the list is not the same, for historical reasons.

As noted previously, the new syntax and rules have been added to accommodate other languages than English. The remainder of this section describes the rationale for these modifications.

In the POSIX locale, a regular expression that starts with a range expression matches a set of strings that are contiguously sorted, but this is not necessarily true in other locales. For example, a French locale might have the following behavior:

```
$ ls
alpha   Alpha   estimé   ESTIMÉ   été   eurêka
$ ls [a-e]*
alpha   Alpha   estimé  eurêka
```

Such disagreements between matching and contiguous sorting are unavoidable because POSIX sorting cannot be implemented in terms of a deterministic finite-state automaton (DFA), but range expressions by design are implementable in terms of DFAs.

Historical implementations used native character order to interpret range expressions. The ISO POSIX-2: 1993 standard instead required collating element order (CEO): the order that collating elements were specified between the **order_start** and **order_end** keywords in the *LC_COLLATE* category of the current locale. CEO had some advantages in portability over the native character order, but it also had some disadvantages:

- CEO could not feasibly be mimicked in user code, leading to inconsistencies between POSIX matchers and matchers in popular user programs like Emacs, *ksh*, and Perl.
- CEO caused range expressions to match accented and capitalized letters contrary to many users' expectations. For example, `"[a-e]"` typically matched both `'E'` and `'á'` but neither `'A'` nor `'é'`.
- CEO was not consistent across implementations. In practice, CEO was often less portable than native character order. For example, it was common for the CEOs of two implementation-supplied locales to disagree, even if both locales were named `"da_DK"`.

Because of these problems, some implementations of regular expressions continued to use native character order. Others used the collation sequence, which is more consistent with sorting than either CEO or native order, but which departs further from the traditional POSIX semantics because it generally requires `"[a-e]"` to match either `'A'` or `'E'` but not both. As a result of this kind of implementation variation, programmers who wanted to write portable regular expressions could not rely on the ISO POSIX-2: 1993 standard guarantees in practice.

While revising the standard, lengthy consideration was given to proposals to attack this problem by adding an API for querying the CEO to allow user-mode matchers, but none of these proposals had implementation experience and none achieved consensus. Leaving the standard alone was also considered, but rejected due to the problems described above.

The current standard leaves unspecified the behavior of a range expression outside the POSIX locale. This makes it clearer that conforming applications should avoid range expressions outside the POSIX locale, and it allows implementations and compatible user-mode matchers to interpret range expressions using native order, CEO, collation sequence, or other, more advanced techniques.

The ISO POSIX-2: 1993 standard required `"[b-a]"` to be an invalid expression in the POSIX locale, but this requirement has been relaxed in this version of the standard so that `"[b-a]"` can instead be treated as a valid expression that does not match any string.

A.9.3.6 BREs Matching Multiple Characters

The limit of nine back-references to subexpressions in the RE is based on the use of a single-digit identifier; increasing this to multiple digits would break historical applications. This does not imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten subexpressions:

```
\(\(\(ab\)*c\)*d\)\(ef\)*\(gh\)\{2\}\(ij\)*\(kl\)*\(mn\)*\(op\)*\(qr\)*
```

The standard developers regarded the common historical behavior, which supported `"\n*"`, but not `"\n\{min,max\}"`, `"\(...\)*"`, or `"\(...\)\{min,max\}"`, as a non-intentional result of a specific implementation, and they supported both duplication and interval expressions following subexpressions and back-references.

The changes to the processing of the back-reference expression remove an unspecified or ambiguous behavior in the Shell and Utilities volume of IEEE Std 1003.1-200x, aligning it with the requirements specified for the *regcomp*( ) expression, and is the result of PASC Interpretation 1003.2-92 #43 submitted for the ISO POSIX-2: 1993 standard.

A.9.3.7 *BRE Precedence*

There is no additional rationale provided for this section.

A.9.3.8 *BRE Expression Anchoring*

Often, the dollar sign is viewed as matching the ending <newline> in text files. This is not strictly true; the <newline> is typically eliminated from the strings to be matched, and the dollar sign matches the terminating null character.

The ability of `'^'`, `'$'`, and `'*'` to be non-special in certain circumstances may be confusing to some programmers, but this situation was changed only in a minor way from historical practice to avoid breaking many historical scripts. Some consideration was given to making the use of the anchoring characters undefined if not escaped and not at the beginning or end of strings. This would cause a number of historical BREs, such as `"2^10"`, `"$HOME"`, and `"$1.35"`, that relied on the characters being treated literally, to become invalid.

However, one relatively uncommon case was changed to allow an extension used on some implementations. Historically, the BREs `"^foo"` and `"\(^foo\)"` did not match the same string, despite the general rule that subexpressions and entire BREs match the same strings. To increase consensus, IEEE Std 1003.1-200x has allowed an extension on some implementations to treat these two cases in the same way by declaring that anchoring *may* occur at the beginning or end of a subexpression. Therefore, portable BREs that require a literal circumflex at the beginning or a dollar sign at the end of a subexpression must escape them. Note that a BRE such as `"a\(^bc\)"` will either match `"a^bc"` or nothing on different systems under the rules.

ERE anchoring has been different from BRE anchoring in all historical systems. An unescaped anchor character has never matched its literal counterpart outside a bracket expression. Some implementations treated `"foo$bar"` as a valid expression that never matched anything; others treated it as invalid. IEEE Std 1003.1-200x mandates the former, valid unmatched behavior.

Some implementations have extended the BRE syntax to add alternation. For example, the subexpression `"\(foo$\|bar\)"` would match either `"foo"` at the end of the string or `"bar"` anywhere. The extension is triggered by the use of the undefined `"\|"` sequence. Because the BRE is undefined for portable scripts, the extending system is free to make other assumptions, such that the `'$'` represents the end-of-line anchor in the middle of a subexpression. If it were not for the extension, the `'$'` would match a literal dollar sign under the rules.

A.9.4 Extended Regular Expressions

As with BREs, the standard developers decided to make the interpretation of escaped ordinary characters undefined.

The right parenthesis is not listed as an ERE special character because it is only special in the context of a preceding left parenthesis. If found without a preceding left parenthesis, the right parenthesis has no special meaning.

The *interval expression*, `"{m,n}"`, has been added to EREs. Historically, the interval expression has only been supported in some ERE implementations. The standard developers estimated that the addition of interval expressions to EREs would not decrease consensus and would also make BREs more of a subset of EREs than in many historical implementations.

It was suggested that, in addition to interval expressions, back-references (`'\n'`) should also be added to EREs. This was rejected by the standard developers as likely to decrease consensus.

In historical implementations, multiple duplication symbols are usually interpreted from left to right and treated as additive. As an example, `"a+*b"` matches zero or more instances of `'a'` followed by a `'b'`. In IEEE Std 1003.1-200x, multiple duplication symbols are undefined; that is, they cannot be relied upon for conforming applications. One reason for this is to provide some scope for future enhancements.

The precedence of operations differs between EREs and those in *lex*; in *lex*, for historical reasons, interval expressions have a lower precedence than concatenation.

A.9.4.1 EREs Matching a Single Character or Collating Element

There is no additional rationale provided for this section.

A.9.4.2 ERE Ordinary Characters

There is no additional rationale provided for this section.

A.9.4.3 ERE Special Characters

There is no additional rationale provided for this section.

A.9.4.4 Periods in EREs

There is no additional rationale provided for this section.

A.9.4.5 ERE Bracket Expression

There is no additional rationale provided for this section.

A.9.4.6 EREs Matching Multiple Characters

There is no additional rationale provided for this section.

A.9.4.7 ERE Alternation

There is no additional rationale provided for this section.

A.9.4.8 ERE Precedence

There is no additional rationale provided for this section.

A.9.4.9 ERE Expression Anchoring

There is no additional rationale provided for this section.

A.9.5 Regular Expression Grammar

The grammars are intended to represent the range of acceptable syntaxes available to conforming applications. There are instances in the text where undefined constructs are described; as explained previously, these allow implementation extensions. There is no intended requirement that an implementation extension must somehow fit into the grammars shown here.

The BRE grammar does not permit L_ANCHOR or R_ANCHOR inside `"\("` and `"\)"` (which implies that `'^'` and `'$'` are ordinary characters). This reflects the semantic limits on the application, as noted in the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.3.8, BRE Expression Anchoring. Implementations are permitted to extend the language to interpret `'^'` and `'$'` as anchors in these locations, and as such, conforming applications cannot use unescaped `'^'` and `'$'` in positions inside `"\("` and `"\)"` that might be interpreted as anchors.

The ERE grammar does not permit several constructs that the Base Definitions volume of IEEE Std 1003.1-200x, Section 9.4.2, ERE Ordinary Characters and the Base Definitions volume of

IEEE Std 1003.1-200x, Section 9.4.3, ERE Special Characters specify as having undefined results:

- ORD_CHAR preceded by `'\'`
- *ERE_dupl_symbol*(s) appearing first in an ERE, or immediately following `'|'`, `'^'`, or `'('`
- `'{'` not part of a valid *ERE_dupl_symbol*
- `'|'` appearing first or last in an ERE, or immediately following `'|'` or `'('`, or immediately preceding `')'`

Implementations are permitted to extend the language to allow these. Conforming applications cannot use such constructs.

A.9.5.1 BRE/ERE Grammar Lexical Conventions

There is no additional rationale provided for this section.

A.9.5.2 RE and Bracket Expression Grammar

The removal of the *Back_open_paren Back_close_paren* option from the *nondupl_RE* specification is the result of PASC Interpretation 1003.2-92 #43 submitted for the ISO POSIX-2:1993 standard. Although the grammar required support for null subexpressions, this section does not describe the meaning of, and historical practice did not support, this construct.

A.9.5.3 ERE Grammar

There is no additional rationale provided for this section.

A.10 Directory Structure and Devices

A.10.1 Directory Structure and Files

A description of the historical **/usr/tmp** was omitted, removing any concept of differences in emphasis between the **/** and **/usr** directories. The descriptions of **/bin**, **/usr/bin**, **/lib**, and **/usr/lib** were omitted because they are not useful for applications. In an early draft, a distinction was made between system and application directory usage, but this was not found to be useful.

The directories **/** and **/dev** are included because the notion of a hierarchical directory structure is key to other information presented elsewhere in IEEE Std 1003.1-200x. In early drafts, it was argued that special devices and temporary files could conceivably be handled without a directory structure on some implementations. For example, the system could treat the characters `"/tmp"` as a special token that would store files using some non-POSIX file system structure. This notion was rejected by the standard developers, who required that all the files in this section be implemented via POSIX file systems.

The **/tmp** directory is retained in IEEE Std 1003.1-200x to accommodate historical applications that assume its availability. Implementations are encouraged to provide suitable directory names in the environment variable *TMPDIR* and applications are encouraged to use the contents of *TMPDIR* for creating temporary files.

The standard files **/dev/null** and **/dev/tty** are required to be both readable and writable to allow applications to have the intended historical access to these files.

The standard file **/dev/console** has been added for alignment with the Single UNIX Specification.

A.10.2 Output Devices and Terminal Types

There is no additional rationale provided for this section.

A.11 General Terminal Interface

If the implementation does not support this interface on any device types, it should behave as if it were being used on a device that is not a terminal device (in most cases *errno* will be set to [ENOTTY] on return from functions defined by this interface). This is based on the fact that many applications are written to run both interactively and in some non-interactive mode, and they adapt themselves at runtime. Requiring that they all be modified to test an environment variable to determine whether they should try to adapt is unnecessary. On a system that provides no general terminal interface, providing all the entry points as stubs that return [ENOTTY] (or an equivalent, as appropriate) has the same effect and requires no changes to the application.

Although the needs of both interface implementors and application developers were addressed throughout IEEE Std 1003.1-200x, this section pays more attention to the needs of the latter. This is because, while many aspects of the programming interface can be hidden from the user by the application developer, the terminal interface is usually a large part of the user interface. Although to some extent the application developer can build missing features or work around inappropriate ones, the difficulties of doing that are greater in the terminal interface than elsewhere. For example, efficiency prohibits the average program from interpreting every character passing through it in order to simulate character erase, line kill, and so on. These functions should usually be done by the operating system, possibly at the interrupt level.

The *tc**( ) functions were introduced as a way of avoiding the problems inherent in the traditional *ioctl*( ) function and in variants of it that were proposed. For example, *tcsetattr*( ) is specified in place of the use of the TCSETA *ioctl*( ) command function. This allows specification of all the arguments in a manner consistent with the ISO C standard unlike the varying third argument of *ioctl*( ), which is sometimes a pointer (to any of many different types) and sometimes an **int**.

The advantages of this new method include:

- It allows strict type checking.
- The direction of transfer of control data is explicit.
- Portable capabilities are clearly identified.
- The need for a general interface routine is avoided.
- Size of the argument is well-defined (there is only one type).

The disadvantages include:

- No historical implementation used the new method.
- There are many small routines instead of one general-purpose one.
- The historical parallel with *fcntl*( ) is broken.

The issue of modem control was excluded from IEEE Std 1003.1-200x on the grounds that:

- It was concerned with setting and control of hardware timers.
- The appropriate timers and settings vary widely internationally.

- Feedback from European computer manufacturers indicated that this facility was not consistent with European needs and that specification of such a facility was not a requirement for portability.

A.11.1 Interface Characteristics

A.11.1.1 Opening a Terminal Device File

There is no additional rationale provided for this section.

A.11.1.2 Process Groups

There is a potential race when the members of the foreground process group on a terminal leave that process group, either by exit or by changing process groups. After the last process exits the process group, but before the foreground process group ID of the terminal is changed (usually by a job-control shell), it would be possible for a new process to be created with its process ID equal to the terminal's foreground process group ID. That process might then become the process group leader and accidentally be placed into the foreground on a terminal that was not necessarily its controlling terminal. As a result of this problem, the controlling terminal is defined to not have a foreground process group during this time.

The cases where a controlling terminal has no foreground process group occur when all processes in the foreground process group either terminate and are waited for or join other process groups via *setpgid*( ) or *setsid*( ). If the process group leader terminates, this is the first case described; if it leaves the process group via *setpgid*( ), this is the second case described (a process group leader cannot successfully call *setsid*( )). When one of those cases causes a controlling terminal to have no foreground process group, it has two visible effects on applications. The first is the value returned by *tcgetpgrp*( ). The second (which occurs only in the case where the process group leader terminates) is the sending of signals in response to special input characters. The intent of IEEE Std 1003.1-200x is that no process group be wrongly identified as the foreground process group by *tcgetpgrp*( ) or unintentionally receive signals because of placement into the foreground.

In 4.3 BSD, the old process group ID continues to be used to identify the foreground process group and is returned by the function equivalent to *tcgetpgrp*( ). In that implementation it is possible for a newly created process to be assigned the same value as a process ID and then form a new process group with the same value as a process group ID. The result is that the new process group would receive signals from this terminal for no apparent reason, and IEEE Std 1003.1-200x precludes this by forbidding a process group from entering the foreground in this way. It would be more direct to place part of the requirement made by the last sentence under *fork*( ), but there is no convenient way for that section to refer to the value that *tcgetpgrp*( ) returns, since in this case there is no process group and thus no process group ID.

One possibility for a conforming implementation is to behave similarly to 4.3 BSD, but to prevent this reuse of the ID, probably in the implementation of *fork*( ), as long as it is in use by the terminal.

Another possibility is to recognize when the last process stops using the terminal's foreground process group ID, which is when the process group lifetime ends, and to change the terminal's foreground process group ID to a reserved value that is never used as a process ID or process group ID. (See the definition of *process group lifetime* in the definitions section.) The process ID can then be reserved until the terminal has another foreground process group.

The 4.3 BSD implementation permits the leader (and only member) of the foreground process group to leave the process group by calling the equivalent of *setpgid*( ) and to later return, expecting to return to the foreground. There are no known application needs for this behavior, and IEEE Std 1003.1-200x neither requires nor forbids it (except that it is forbidden for session leaders) by leaving it unspecified.

A.11.1.3 The Controlling Terminal

IEEE Std 1003.1-200x does not specify a mechanism by which to allocate a controlling terminal. This is normally done by a system utility (such as *getty*) and is considered an administrative feature outside the scope of IEEE Std 1003.1-200x.

Historical implementations allocate controlling terminals on certain *open*( ) calls. Since *open*( ) is part of POSIX.1, its behavior had to be dealt with. The traditional behavior is not required because it is not very straightforward or flexible for either implementations or applications. However, because of its prevalence, it was not practical to disallow this behavior either. Thus, a mechanism was standardized to ensure portable, predictable behavior in *open*( ).

Some historical implementations deallocate a controlling terminal on the last system-wide close. This behavior in neither required nor prohibited. Even on implementations that do provide this behavior, applications generally cannot depend on it due to its system-wide nature.

A.11.1.4 Terminal Access Control

The access controls described in this section apply only to a process that is accessing its controlling terminal. A process accessing a terminal that is not its controlling terminal is effectively treated the same as a member of the foreground process group. While this may seem unintuitive, note that these controls are for the purpose of job control, not security, and job control relates only to a process' controlling terminal. Normal file access permissions handle security.

If the process calling *read*( ) or *write*( ) is in a background process group that is orphaned, it is not desirable to stop the process group, as it is no longer under the control of a job control shell that could put it into foreground again. Accordingly, calls to *read*( ) or *write*( ) functions by such processes receive an immediate error return. This is different from 4.2 BSD, which kills orphaned processes that receive terminal stop signals. |

The foreground / background / orphaned process group check performed by the terminal driver must be repeatedly performed until the calling process moves into the foreground or until the process group of the calling process becomes orphaned. That is, when the terminal driver determines that the calling process is in the background and should receive a job control signal, it sends the appropriate signal (SIGTTIN or SIGTTOU) to every process in the process group of the calling process and then it allows the calling process to immediately receive the signal. The latter is typically performed by blocking the process so that the signal is immediately noticed. Note, however, that after the process finishes receiving the signal and control is returned to the driver, the terminal driver must reexecute the foreground / background / orphaned process group check. The process may still be in the background, either because it was continued in the background by a job-control shell, or because it caught the signal and did nothing.

The terminal driver repeatedly performs the foreground / background / orphaned process group checks whenever a process is about to access the terminal. In the case of *write*( ) or the control *tc**( ) functions, the check is performed at the entry of the function. In the case of *read*( ), the check is performed not only at the entry of the function, but also after blocking the process to wait for input characters (if necessary). That is, once the driver has determined that the process calling the *read*( ) function is in the foreground, it attempts to retrieve characters from the input queue. If the queue is empty, it blocks the process waiting for characters. When characters are available and control is returned to the driver, the terminal driver must return to the repeated foreground / background / orphaned process group check again. The process may have moved from the foreground to the background while it was blocked waiting for input characters.

A.11.1.5 Input Processing and Reading Data

There is no additional rationale provided for this section.

A.11.1.6 Canonical Mode Input Processing

The term *character* is intended here. ERASE should erase the last character, not the last byte. In the case of multi-byte characters, these two may be different.

4.3 BSD has a WERASE character that erases the last ‘‘word’’ typed (but not any preceding <blank>s or <tab>s). A word is defined as a sequence of non-<blank>s, with <tab>s counted as <blank>s. Like ERASE, WERASE does not erase beyond the beginning of the line. This WERASE feature has not been specified in POSIX.1 because it is difficult to define in the international environment. It is only useful for languages where words are delimited by <blank>s. In some ideographic languages, such as Japanese and Chinese, words are not delimited at all. The WERASE character should presumably take one back to the beginning of a sentence in those cases; practically, this means it would not get much use for those languages.

It should be noted that there is a possible inherent deadlock if the application and implementation conflict on the value of MAX_CANON. With ICANON set (if IXOFF is enabled) and more than MAX_CANON characters transmitted without a <linefeed>, transmission will be stopped, the <linefeed> (or <carriage-return> when ICRLF is set) will never arrive, and the *read*( ) will never be satisfied.

An application should not set IXOFF if it is using canonical mode unless it knows that (even in the face of a transmission error) the conditions described previously cannot be met or unless it is prepared to deal with the possible deadlock in some other way, such as timeouts.

It should also be noted that this can be made to happen in non-canonical mode if the trigger value for sending IXOFF is less than VMIN and VTIME is zero.

A.11.1.7 Non-Canonical Mode Input Processing

Some points to note about MIN and TIME:

1. The interactions of MIN and TIME are not symmetric. For example, when MIN>0 and TIME=0, TIME has no effect. However, in the opposite case where MIN=0 and TIME>0, both MIN and TIME play a role in that MIN is satisfied with the receipt of a single character.
2. Also note that in case A (MIN>0, TIME>0), TIME represents an inter-character timer, while in case C (MIN=0, TIME>0), TIME represents a read timer.

These two points highlight the dual purpose of the MIN/TIME feature. Cases A and B, where MIN>0, exist to handle burst-mode activity (for example, file transfer programs) where a program would like to process at least MIN characters at a time. In case A, the inter-character timer is activated by a user as a safety measure; in case B, it is turned off.

Cases C and D exist to handle single-character timed transfers. These cases are readily adaptable to screen-based applications that need to know if a character is present in the input queue before refreshing the screen. In case C, the read is timed; in case D, it is not.

Another important note is that MIN is always just a minimum. It does not denote a record length. That is, if a program does a read of 20 bytes, MIN is 10, and 25 characters are present, 20 characters shall be returned to the user. In the special case of MIN=0, this still applies: if more than one character is available, they all will be returned immediately.

A.11.1.8 Writing Data and Output Processing

There is no additional rationale provided for this section.

A.11.1.9 Special Characters

There is no additional rationale provided for this section.

A.11.1.10 Modem Disconnect

There is no additional rationale provided for this section.

A.11.1.11 Closing a Terminal Device File

IEEE Std 1003.1-200x does not specify that a *close*( ) on a terminal device file include the equivalent of a call to *tcflow*(*fd*,TCOON).

An implementation that discards output at the time *close*( ) is called after reporting the return value to the *write*( ) call that data was written does not conform with IEEE Std 1003.1-200x. An application has functions such as *tcdrain*( ), *tcflush*( ), and *tcflow*( ) available to obtain the detailed behavior it requires with respect to flushing of output.

At the time of the last close on a terminal device, an application relinquishes any ability to exert flow control via *tcflow*( ).

A.11.2 Parameters that Can be Set

A.11.2.1 The termios Structure

This structure is part of an interface that, in general, retains the historic grouping of flags. Although a more optimal structure for implementations may be possible, the degree of change to applications would be significantly larger.

A.11.2.2 Input Modes

Some historical implementations treated a long break as multiple events, as many as one per character time. The wording in POSIX.1 explicitly prohibits this.

Although the ISTRIP flag is normally superfluous with today's terminal hardware and software, it is historically supported. Therefore, applications may be using ISTRIP, and there is no technical problem with supporting this flag. Also, applications may wish to receive only 7-bit input bytes and may not be connected directly to the hardware terminal device (for example, when a connection traverses a network).

Also, there is no requirement in general that the terminal device ensures that high-order bits beyond the specified character size are cleared. ISTRIP provides this function for 7-bit characters, which are common.

In dealing with multi-byte characters, the consequences of a parity error in such a character, or in an escape sequence affecting the current character set, are beyond the scope of POSIX.1 and are best dealt with by the application processing the multi-byte characters.

A.11.2.3 Output Modes

POSIX.1 does not describe postprocessing of output to a terminal or detailed control of that from a conforming application. (That is, translation of <newline> to <carriage-return> followed by <linefeed> or <tab> processing.) There is nothing that a conforming application should do to its output for a terminal because that would require knowledge of the operation of the terminal. It is the responsibility of the operating system to provide postprocessing appropriate to the output device, whether it is a terminal or some other type of device.

Extensions to POSIX.1 to control the type of postprocessing already exist and are expected to continue into the future. The control of these features is primarily to adjust the interface between the system and the terminal device so the output appears on the display correctly. This should be set up before use by any application.

In general, both the input and output modes should not be set absolutely, but rather modified from the inherited state.

A.11.2.4 Control Modes

This section could be misread that the symbol ‘‘CSIZE’’ is a title in the **termios** *c_cflag* field . Although it does serve that function, it is also a required symbol, as a literal reading of POSIX.1 (and the caveats about typography) would indicate.

A.11.2.5 Local Modes

Non-canonical mode is provided to allow fast bursts of input to be read efficiently while still allowing single-character input.

The ECHONL function historically has been in many implementations. Since there seems to be no technical problem with supporting ECHONL, it is included in POSIX.1 to increase consensus.

The alternate behavior possible when ECHOK or ECHOE are specified with ICANON is permitted as a compromise depending on what the actual terminal hardware can do. Erasing characters and lines is preferred, but is not always possible.

A.11.2.6 Special Control Characters

Permitting VMIN and VTIME to overlap with VEOF and VEOL was a compromise for historical implementations. Only when backwards-compatibility of object code is a serious concern to an implementor should an implementation continue this practice. Correct applications that work with the overlap (at the source level) should also work if it is not present, but not the reverse.

A.12 Utility Conventions

A.12.1 Utility Argument Syntax

The standard developers considered that recent trends toward diluting the SYNOPSIS sections of historical reference pages to the equivalent of:

```
command [options][operands]
```

were a disservice to the reader. Therefore, considerable effort was placed into rigorous definitions of all the command line arguments and their interrelationships. The relationships depicted in the synopses are normative parts of IEEE Std 1003.1-200x; this information is sometimes repeated in textual form, but that is only for clarity within context.

The use of ''undefined'' for conflicting argument usage and for repeated usage of the same option is meant to prevent conforming applications from using conflicting arguments or repeated options unless specifically allowed (as is the case with *ls*, which allows simultaneous, repeated use of the **−C**, **−l**, and **−1** options). Many historical implementations will tolerate this usage, choosing either the first or the last applicable argument. This tolerance can continue, but conforming applications cannot rely upon it. (Other implementations may choose to print usage messages instead.)

The use of ''undefined'' for conflicting argument usage also allows an implementation to make reasonable extensions to utilities where the implementor considers mutually-exclusive options according to IEEE Std 1003.1-200x to have a sensible meaning and result.

IEEE Std 1003.1-200x does not define the result of a command when an option-argument or operand is not followed by ellipses and the application specifies more than one of that option-argument or operand. This allows an implementation to define valid (although non-standard) behavior for the utility when more than one such option or operand is specified.

Allowing <blank>s after an option (that is, placing an option and its option-argument into separate argument strings) when IEEE Std 1003.1-200x does not require it encourages portability of users, while still preserving backwards-compatibility of scripts. Inserting <blank>s between the option and the option-argument is preferred; however, historical usage has not been consistent in this area; therefore, <blank>s are required to be handled by all implementations, but implementations are also allowed to handle the historical syntax. Another justification for selecting the multiple-argument method was that the single-argument case is inherently ambiguous when the option-argument can legitimately be a null string.

IEEE Std 1003.1-200x explicitly states that digits are permitted as operands and option-arguments. The lower and upper bounds for the values of the numbers used for operands and option-arguments were derived from the ISO C standard values for {LONG_MIN} and {LONG_MAX}. The requirement on the standard utilities is that numbers in the specified range do not cause a syntax error, although the specification of a number need not be semantically correct for a particular operand or option-argument of a utility. For example, the specification of:

```
dd obs=3000000000
```

would yield undefined behavior for the application and could be a syntax error because the number 3 000 000 000 is outside of the range −2 147 483 647 to +2 147 483 647. On the other hand:

```
dd obs=2000000000
```

may cause some error, such as ''blocksize too large'', rather than a syntax error.

A.12.2 Utility Syntax Guidelines

This section is based on the rules listed in the SVID. It was included for two reasons:

1. The individual utility descriptions in the Shell and Utilities volume of IEEE Std 1003.1-200x, Chapter 4, Utilities needed a set of common (although not universal) actions on which they could anchor their descriptions of option and operand syntax. Most of the standard utilities actually do use these guidelines, and many of their historical implementations use the *getopt*( ) function for their parsing. Therefore, it was simpler to cite the rules and merely identify exceptions.

2. Writers of conforming applications need suggested guidelines if the POSIX community is to avoid the chaos of historical UNIX system command syntax. |

It is recommended that all *future* utilities and applications use these guidelines to enhance ‘‘user portability’’. The fact that some historical utilities could not be changed (to avoid breaking historical applications) should not deter this future goal.

The voluntary nature of the guidelines is highlighted by repeated uses of the word *should* throughout. This usage should not be misinterpreted to imply that utilities that claim conformance in their OPTIONS sections do not always conform.

Guidelines 1 and 2 are offered as guidance for locales using Latin alphabets. No recommendations are made by IEEE Std 1003.1-200x concerning utility naming in other locales.

In the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1, Simple Commands, it is further stated that a command used in the Shell Command Language cannot be named with a trailing colon.

Guideline 3 was changed to allow alphanumeric characters (letters and digits) from the character set to allow compatibility with historical usage. Historical practice allows the use of digits wherever practical, and there are no portability issues that would prohibit the use of digits. In fact, from an internationalization viewpoint, digits (being non-language-dependent) are preferable over letters (a **–2** is intuitively self-explanatory to any user, while in the **–f** *filename* the letter `'f'` is a mnemonic aid only to speakers of Latin-based languages where ‘‘filename’’ happens to translate to a word that begins with `'f'`. Since guideline 3 still retains the word ‘‘single’’, multi-digit options are not allowed. Instances of historical utilities that used them have been marked obsolescent, with the numbers being changed from option names to option-arguments.

It was difficult to achieve a satisfactory solution to the problem of name space in option characters. When the standard developers desired to extend the historical *cc* utility to accept ISO C standard programs, they found that all of the portable alphabet was already in use by various vendors. Thus, they had to devise a new name, *c89*, rather than something like *cc* **–X**. There were suggestions that implementors be restricted to providing extensions through various means (such as using a plus sign as the option delimiter or using option characters outside the alphanumeric set) that would reserve all of the remaining alphanumeric characters for future POSIX standards. These approaches were resisted because they lacked the historical style of UNIX systems. Furthermore, if a vendor-provided option should become commonly used in the industry, it would be a candidate for standardization. It would be desirable to standardize such a feature using historical practice for the syntax (the semantics can be standardized with any syntax). This would not be possible if the syntax was one reserved for the vendor. However, since the standardization process may lead to minor changes in the semantics, it may prove to be better for a vendor to use a syntax that will not be affected by standardization.

Guideline 8 includes the concept of comma-separated lists in a single argument. It is up to the utility to parse such a list itself because *getopt*( ) just returns the single string. This situation was retained so that certain historical utilities would not violate the guidelines. Applications preparing for international use should be aware of an occasional problem with comma-separated lists: in some locales, the comma is used as the radix character. Thus, if an application is preparing operands for a utility that expects a comma-separated lists, it should avoid generating non-integer values through one of the means that is influenced by setting the *LC_NUMERIC* variable (such as *awk*, *bc*, *printf*, or *printf*( )).

Applications calling any utility with a first operand starting with `'−'` should usually specify −−, as indicated by Guideline 10, to mark the end of the options. This is true even if the SYNOPSIS in the Shell and Utilities volume of IEEE Std 1003.1-200x does not specify any options; implementations may provide options as extensions to the Shell and Utilities volume of IEEE Std 1003.1-200x. The standard utilities that do not support Guideline 10 indicate that fact in the OPTIONS section of the utility description.

Guideline 11 was modified to clarify that the order of different options should not matter relative to one another. However, the order of repeated options that also have option-arguments may be significant; therefore, such options are required to be interpreted in the order that they are specified. The *make* utility is an instance of a historical utility that uses repeated options in which the order is significant. Multiple files are specified by giving multiple instances of the −**f** option; for example:

```
make −f common_header −f specific_rules target
```

Guideline 13 does not imply that all of the standard utilities automatically accept the operand `'−'` to mean standard input or output, nor does it specify the actions of the utility upon encountering multiple `'−'` operands. It simply says that, by default, `'−'` operands are not used for other purposes in the file reading or writing (but not when using *stat*, *unlink*, *touch*, and so on) utilities. All information concerning actual treatment of the `'−'` operand is found in the individual utility sections.

An area of concern was that as implementations mature, implementation-defined utilities and implementation-defined utility options will result. The idea was expressed that there needed to be a standard way, say an environment variable or some such mechanism, to identify implementation-defined utilities separately from standard utilities that may have the same name. It was decided that there already exist several ways of dealing with this situation and that it is outside of the POSIX.2 scope to attempt to standardize in the area of non-standard items. A method that exists on some historical implementations is the use of the so-called **/local/bin** or **/usr/local/bin** directory to separate local or additional copies or versions of utilities. Another method that is also used is to isolate utilities into completely separate domains. Still another method to ensure that the desired utility is being used is to request the utility by its full pathname. There are many approaches to this situation; the examples given above serve to illustrate that there is more than one.

A.13 Headers

A.13.1 Format of Entries

Each header reference page has a common layout of sections describing the interface. This layout is similar to the manual page or ‘‘man’’ page format shipped with most UNIX systems, and each header has sections describing the SYNOPSIS and DESCRIPTION. These are the two sections that relate to conformance.

Additional sections are informative, and add considerable information for the application developer. APPLICATION USAGE sections provide additional caveats, issues, and recommendations to the developer. RATIONALE sections give additional information on the decisions made in defining the interface.

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

The CHANGE HISTORY section describes when the interface was introduced, and how it has changed.

Option labels and margin markings in the page can be useful in guiding the application developer.

Rationale (Informative)

Part B:

System Interfaces

The Open Group
The Institute of Electrical and Electronics Engineers, Inc.

Appendix B

Rationale for System Interfaces

B.1 Introduction

B.1.1 Scope

Refer to Section A.1.1 (on page 3293).

B.1.2 Conformance

Refer to Section A.2 (on page 3299).

B.1.3 Normative References

There is no additional rationale provided for this section.

B.1.4 Change History

The change history is provided as an informative section, to track changes from previous issues of IEEE Std 1003.1-200x.

The following sections describe changes made to the System Interfaces volume of IEEE Std 1003.1-200x since Issue 5 of the base document. The CHANGE HISTORY section for each entry details the technical changes that have been made to that entry since Issue 5. Changes between earlier issues of the base document and Issue 5 are not included.

The change history between Issue 5 and Issue 6 also lists the changes since the ISO POSIX-1: 1996 standard.

Changes from Issue 5 to Issue 6 (IEEE Std 1003.1-200x)

The following list summarizes the major changes that were made in the System Interfaces volume of IEEE Std 1003.1-200x from Issue 5 to Issue 6:

- This volume of IEEE Std 1003.1-200x is extensively revised so it can be both an IEEE POSIX Standard and an Open Group Technical Standard.
- The POSIX System Interfaces requirements incorporate support of FIPS 151-2.
- The POSIX System Interfaces requirements are updated to align with some features of the Single UNIX Specification.
- A RATIONALE section is added to each reference page.
- Networking interfaces from the XNS, Issue 5.2 specification are incorporated.
- IEEE Std 1003.1d-1999 is incorporated.
- IEEE Std 1003.1j-2000 is incorporated.
- IEEE Std 1003.1q-2000 is incorporated.
- IEEE P1003.1a draft standard is incorporated.

- Existing functionality is aligned with the ISO / IEC 9899: 1999 standard.
- New functionality from the ISO / IEC 9899: 1999 standard is incorporated.
- IEEE PASC Interpretations are applied.
- The Open Group corrigenda and resolutions are applied.

New Features in Issue 6

The functions first introduced in Issue 6 (over the Issue 5 Base document) are listed in the table below:

New Functions in Issue 6		
acosf( )	*catanhl*( )	*cprojf*( )
acoshf( )	*catanl*( )	*cprojl*( )
acoshl( )	*cbrtf*( )	*creal*( )
acosl( )	*cbrtl*( )	*crealf*( )
asinf( )	*ccos*( )	*creall*( )
asinhf( )	*ccosf*( )	*csin*( )
asinhl( )	*ccosh*( )	*csinf*( )
asinl( )	*ccoshf*( )	*csinh*( )
atan2f( )	*ccoshl*( )	*csinhf*( )
atan2l( )	*ccosl*( )	*csinhl*( )
atanf( )	*ceilf*( )	*csinl*( )
atanhf( )	*ceill*( )	*csqrt*( )
atanhl( )	*cexp*( )	*csqrtf*( )
atanl( )	*cexpf*( )	*csqrtl*( )
atoll( )	*cexpl*( )	*ctan*( )
cabs( )	*cimag*( )	*ctanf*( )
cabsf( )	*cimagf*( )	*ctanh*( )
cabsl( )	*cimagl*( )	*ctanhf*( )
cacos( )	*clock_getcpuclockid*( )	*ctanhl*( )
cacosf( )	*clock_nanosleep*( )	*ctanl*( )
cacosh( )	*clog*( )	*erfcf*( )
cacoshf( )	*clogf*( )	*erfcl*( )
cacoshl( )	*clogl*( )	*erff*( )
cacosl( )	*conj*( )	*erfl*( )
carg( )	*conjf*( )	*exp2*( )
cargf( )	*conjl*( )	*exp2f*( )
cargl( )	*copysign*( )	*exp2l*( )
casin( )	*copysignf*( )	*expf*( )
casinf( )	*copysignl*( )	*expl*( )
casinh( )	*cosf*( )	*expm1f*( )
casinhf( )	*coshf*( )	*expm1l*( )
casinhl( )	*coshl*( )	*fabsf*( )
casinl( )	*cosl*( )	*fabsl*( )
catan( )	*cpow*( )	*fdim*( )
catanf( )	*cpowf*( )	*fdimf*( )
catanh( )	*cpowl*( )	*fdiml*( )
catanhf( )	*cproj*( )	*feclearexcept*( )

New Functions in Issue 6

fegetenv( )	*ldexpl*( )	*posix_fallocate*( )
fegetexceptflag( )	*lgammaf*( )	*posix_madvise*( )
fegetround( )	*lgammal*( )	*posix_mem_offset*( )
feholdexcept( )	*llabs*( )	*posix_memalign*( )
feraiseexcept( )	*lldiv*( )	*posix_openpt*( )
fesetenv( )	*llrint*( )	*posix_spawn*( )
fesetexceptflag( )	*llrintf*( )	*posix_spawn_file_actions_addclose*( )
fesetround( )	*llrintl*( )	*posix_spawn_file_actions_adddup2*( )
fetestexcept( )	*llround*( )	*posix_spawn_file_actions_addopen*( )
feupdateenv( )	*llroundf*( )	*posix_spawn_file_actions_destroy*( )
floorf( )	*llroundl*( )	*posix_spawn_file_actions_init*( )
floorl( )	*log10f*( )	*posix_spawnattr_destroy*( )
fma( )	*log10l*( )	*posix_spawnattr_getflags*( )
fmaf( )	*log1pf*( )	*posix_spawnattr_getpgroup*( )
fmal( )	*log1pl*( )	*posix_spawnattr_getschedparam*( )
fmax( )	*log2*( )	*posix_spawnattr_getschedpolicy*( )
fmaxf( )	*log2f*( )	*posix_spawnattr_getsigdefault*( )
fmaxl( )	*log2l*( )	*posix_spawnattr_getsigmask*( )
fmin( )	*logbf*( )	*posix_spawnattr_init*( )
fminf( )	*logbl*( )	*posix_spawnattr_setflags*( )
fminl( )	*logf*( )	*posix_spawnattr_setpgroup*( )
fmodf( )	*logl*( )	*posix_spawnattr_setschedparam*( )
fmodl( )	*lrint*( )	*posix_spawnattr_setschedpolicy*( )
fpclassify( )	*lrintf*( )	*posix_spawnattr_setsigdefault*( )
frexpf( )	*lrintl*( )	*posix_spawnattr_setsigmask*( )
frexpl( )	*lround*( )	*posix_spawnp*( )
hypotf( )	*lroundf*( )	*posix_trace_attr_destroy*( )
hypotl( )	*lroundl*( )	*posix_trace_attr_getclockres*( )
ilogbf( )	*modff*( )	*posix_trace_attr_getcreatetime*( )
ilogbl( )	*modfl*( )	*posix_trace_attr_getgenversion*( )
imaxabs( )	*mq_timedreceive*( )	*posix_trace_attr_getinherited*( )
imaxdiv( )	*mq_timedsend*( )	*posix_trace_attr_getlogfullpolicy*( )
isblank( )	*nan*( )	*posix_trace_attr_getlogsize*( )
isfinite( )	*nanf*( )	*posix_trace_attr_getmaxdatasize*( )
isgreater( )	*nanl*( )	*posix_trace_attr_getmaxsystemeventsize*( )
isgreaterequal( )	*nearbyint*( )	*posix_trace_attr_getmaxusereventsize*( )
isinf( )	*nearbyintf*( )	*posix_trace_attr_getname*( )
isless( )	*nearbyintl*( )	*posix_trace_attr_getstreamfullpolicy*( )
islessequal( )	*nextafterf*( )	*posix_trace_attr_getstreamsize*( )
islessgreater( )	*nextafterl*( )	*posix_trace_attr_init*( )
isnormal( )	*nexttoward*( )	*posix_trace_attr_setinherited*( )
isunordered( )	*nexttowardf*( )	*posix_trace_attr_setlogfullpolicy*( )
iswblank( )	*nexttowardl*( )	*posix_trace_attr_setlogsize*( )
ldexpf( )	*posix_fadvise*( )	*posix_trace_create*( )

New Functions in Issue 6

posix_trace_attr_setmaxdatasize( )
posix_trace_attr_setname( )
posix_trace_attr_setstreamfullpolicy( )
posix_trace_attr_setstreamsize( )
posix_trace_clear( )
posix_trace_close( )
posix_trace_create_withlog( )
posix_trace_event( )
posix_trace_eventid_equal( )
posix_trace_eventid_get_name( )
posix_trace_eventid_open( )
posix_trace_eventset_add( )
posix_trace_eventset_del( )
posix_trace_eventset_empty( )
posix_trace_eventset_fill( )
posix_trace_eventset_ismember( )
posix_trace_eventtypelist_getnext_id( )
posix_trace_eventtypelist_rewind( )
posix_trace_flush( )
posix_trace_get_attr( )
posix_trace_get_filter( )
posix_trace_get_status( )
posix_trace_getnext_event( )
posix_trace_open( )
posix_trace_rewind( )
posix_trace_set_filter( )
posix_trace_shutdown( )
posix_trace_start( )
posix_trace_stop( )
posix_trace_timedgetnext_event( )
posix_trace_trid_eventid_open( )
posix_trace_trygetnext_event( )
posix_typed_mem_get_info( )
posix_typed_mem_open( )
powf( )
powl( )
pselect( )
pthread_attr_getstack( )
pthread_attr_setstack( )
pthread_barrier_destroy( )
pthread_barrier_init( )
pthread_barrier_wait( )
pthread_barrierattr_destroy( )
pthread_barrierattr_getpshared( )
pthread_barrierattr_init( )
pthread_barrierattr_setpshared( )
pthread_condattr_getclock( )
pthread_condattr_setclock( )
pthread_getcpuclockid( )
pthread_mutex_timedlock( )
pthread_rwlock_timedrdlock( )
pthread_rwlock_timedwrlock( )
pthread_schedsetprio( )
pthread_spin_destroy( )
pthread_spin_init( )
pthread_spin_lock( )
pthread_spin_trylock( )
pthread_spin_unlock( )
remainderf( )
remainderl( )
remquo( )
remquof( )
remquol( )
rintf( )
rintl( )
round( )
roundf( )
roundl( )
scalbln( )
scalblnf( )
scalblnl( )
scalbn( )
scalbnf( )
scalbnl( )
sem_timedwait( )
setegid( )
setenv( )
seteuid( )
signbit( )
sinf( )
sinhf( )
sinhl( )
sinl( )
sqrtf( )
sqrtl( )
strerror_r( )
stroull( )
strtoimax( )
strtoll( )
strtoumax( )
tanf( )
tanhf( )
tanhl( )
tanl( )
tgamma( )
tgammaf( )
tgammal( )
trunc( )
truncf( )
truncl( )
unsetenv( )
vfprintf( )
vfscanf( )
vfwscanf( )
vprintf( )
vscanf( )
vsnprintf( )
vsprintf( )
vsscanf( )
vswscanf( )
vwscanf( )
wcstoimax( )
wcstoll( )
wcstoull( )
wcstoumax( )

The following new headers are introduced in Issue 6:

New Headers in Issue 6		
<complex.h>	**<spawn.h>**	**<tgmath.h>**
<fenv.h>	**<stdbool.h>**	**<trace.h>**
<net/if.h>	**<stdint.h>**	

The following table lists the functions and symbols from the XSI extension. These are new since the ISO POSIX-1:1996 standard.

New XSI Functions and Symbols in Issue 6

_longjmp()
_setjmp()
_tolower()
_toupper()
a64l()
basename()
bcmp()
bcopy()
bzero()
catclose()
catgets()
catopen()
closelog()
crypt()
daylight
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlclose()
dlerror()
dlopen()
dlsym()
drand48()
ecvt()
encrypt()
endgrent()
endpwent()
endutxent()
erand48()
fchdir()
fcvt()
ffs()
fmtmsg()
fstatvfs()
ftime()
ftok()
ftw()
gcvt()
getcontext()
getdate()
getgrent()
gethostid()
getitimer()
getpgid()
getpmsg()
getpriority()
getpwent()
getrlimit()
getrusage()
getsid()
getsubopt()
gettimeofday()
getutxent()
getutxid()
getutxline()
getwd()
grantpt()
hcreate()
hdestroy()
hsearch()
iconv()
iconv_close()
iconv_open()
index()
initstate()
insque()
isascii()
jrand48()
killpg()
l64a()
lchown()
lcong48()
lfind()
lockf()
lrand48()
lsearch()
makecontext()
memccpy()
mknod()
mkstemp()
mktemp()
mrand48()
msgctl()
msgget()
msgrcv()
msgsnd()
nftw()
nice()
nl_langinfo()
nrand48()
openlog()
poll()
pread()
pthread_attr_getguardsize()
pthread_attr_setguardsize()
pthread_attr_setstack()
pthread_getconcurrency()
pthread_mutexattr_gettype()
pthread_mutexattr_settype()
pthread_rwlockattr_init()
pthread_rwlockattr_setpshared()
pthread_setconcurrency()
ptsname()
putenv()
pututxline()
pwrite()
random()
readv()
realpath()
remque()
rindex()
seed48()
seekdir()
semctl()
semget()
semop()
setcontext()
setgrent()
setitimer()
setkey()
setlogmask()
setpgrp()
setpriority()
setpwent()
setregid()
setreuid()
setrlimit()
setstate()
setutxent()
shmat()
shmctl()
shmdt()
shmget()
sigaltstack()
sighold()
sigignore()
siginterrupt()
sigpause()
sigrelse()
sigset()
srand48()
srandom()
statvfs()
strcasecmp()
strdup()
strfmon()
strncasecmp()
strptime()
swab()
swapcontext()
sync()
syslog()
tcgetsid()
tdelete()
telldir()
tempnam()
tfind()
timezone
toascii()
truncate()
tsearch()
twalk()
ulimit()
unlockpt()
utimes()
waitid()
wcswcs()
wcswidth()
wcwidth()
writev()

The following table lists the headers from the XSI extension. These are new since the ISO POSIX-1:1996 standard.

New XSI Headers in Issue 6		
<cpio.h>	**<poll.h>**	**<sys/statvfs.h>**
<dlfcn.h>	**<search.h>**	**<sys/time.h>**
<fmtmsg.h>	**<strings.h>**	**<sys/timeb.h>**
<ftw.h>	**<stropts.h>**	**<sys/uio.h>**
<iconv.h>	**<sys/ipc.h>**	**<syslog.h>**
<langinfo.h>	**<sys/mman.h>**	**<ucontext.h>**
<libgen.h>	**<sys/msg.h>**	**<ulimit.h>**
<monetary.h>	**<sys/resource.h>**	**<utmpx.h>**
<ndbm.h>	**<sys/sem.h>**	
<nl_types.h>	**<sys/shm.h>**	

B.1.5 Terminology

Refer to Section A.1.4 (on page 3295).

B.1.6 Definitions

Refer to Section A.3 (on page 3302).

B.1.7 Relationship to Other Formal Standards

There is no additional rationale provided for this section.

B.1.8 Portability

Refer to Section A.1.5 (on page 3298).

B.1.8.1 Codes

Refer to Section A.1.5.1 (on page 3298).

B.1.9 Format of Entries

Each system interface reference page has a common layout of sections describing the interface. This layout is similar to the manual page or ‘‘man’’ page format shipped with most UNIX systems, and each header has sections describing the SYNOPSIS, DESCRIPTION, RETURN VALUE, and ERRORS. These are the four sections that relate to conformance.

Additional sections are informative, and add considerable information for the application developer. EXAMPLES sections provide example usage. APPLICATION USAGE sections provide additional caveats, issues, and recommendations to the developer. RATIONALE sections give additional information on the decisions made in defining the interface.

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

The CHANGE HISTORY section describes when the interface was introduced, and how it has changed.

Option labels and margin markings in the page can be useful in guiding the application developer.

B.2 General Information

B.2.1 Use and Implementation of Functions

The information concerning the use of functions was adapted from a description in the ISO C standard. Here is an example of how an application program can protect itself from functions that may or may not be macros, rather than true functions:

The *atoi*( ) function may be used in any of several ways:

- By use of its associated header (possibly generating a macro expansion):

  ```
  #include <stdlib.h>
  /* ... */
  i = atoi(str);
  ```

- By use of its associated header (assuredly generating a true function call):

  ```
  #include <stdlib.h>
  #undef atoi
  /* ... */
  i = atoi(str);
  ```

  or:

  ```
  #include <stdlib.h>
  /* ... */
  i = (atoi) (str);
  ```

- By explicit declaration:

  ```
  extern int atoi (const char *);
  /* ... */
  i = atoi(str);
  ```

- By implicit declaration:

  ```
  /* ... */
  i = atoi(str);
  ```

  (Assuming no function prototype is in scope. This is not allowed by the ISO C standard for functions with variable arguments; furthermore, parameter type conversion ‘‘widening’’ is subject to different rules in this case.)

Note that the ISO C standard reserves names starting with `'_'` for the compiler. Therefore, the compiler could, for example, implement an intrinsic, built-in function *_asm_builtin_atoi*( ), which it recognized and expanded into inline assembly code. Then, in **<stdlib.h>**, there could be the following:

```
#define atoi(X) _asm_builtin_atoi(X)
```

The user’s ‘‘normal’’ call to *atoi*( ) would then be expanded inline, but the implementor would also be required to provide a callable function named *atoi*( ) for use when the application requires it; for example, if its address is to be stored in a function pointer variable.

B.2.2 The Compilation Environment

B.2.2.1 POSIX.1 Symbols

This and the following section address the issue of ‘‘name space pollution’’. The ISO C standard requires that the name space beyond what it reserves not be altered except by explicit action of the application writer. This section defines the actions to add the POSIX.1 symbols for those headers where both the ISO C standard and POSIX.1 need to define symbols, and also where the XSI Extension extends the base standard.

When headers are used to provide symbols, there is a potential for introducing symbols that the application writer cannot predict. Ideally, each header should only contain one set of symbols, but this is not practical for historical reasons. Thus, the concept of feature test macros is included. Two feature test macros are explicitly defined by IEEE Std 1003.1-200x; it is expected that future revisions may add to this.

Note: Feature test macros allow an application to announce to the implementation its desire to have certain symbols and prototypes exposed. They should not be confused with the version test macros and constants for options in **<unistd.h>** which are the implementation’s way of announcing functionality to the application.

It is further intended that these feature test macros apply only to the headers specified by IEEE Std 1003.1-200x. Implementations are expressly permitted to make visible symbols not specified by IEEE Std 1003.1-200x, within both POSIX.1 and other headers, under the control of feature test macros that are not defined by IEEE Std 1003.1-200x.

The _POSIX_C_SOURCE Feature Test Macro

Since _POSIX_SOURCE specified by the POSIX.1-1990 standard did not have a value associated with it, the _POSIX_C_SOURCE macro replaces it, allowing an application to inform the system of the revision of the standard to which it conforms. This symbol will allow implementations to support various revisions of IEEE Std 1003.1-200x simultaneously. For instance, when either _POSIX_SOURCE is defined or _POSIX_C_SOURCE is defined as 1, the system should make visible the same name space as permitted and required by the POSIX.1-1990 standard. When _POSIX_C_SOURCE is defined, the state of _POSIX_SOURCE is completely irrelevant.

It is expected that C bindings to future POSIX standards will define new values for _POSIX_C_SOURCE, with each new value reserving the name space for that new standard, plus all earlier POSIX standards.

The _XOPEN_SOURCE Feature Test Macro

The feature test macro _XOPEN_SOURCE is provided as the announcement mechanism for the application that it requires functionality from the Single UNIX Specification. _XOPEN_SOURCE must be defined to the value 600 before the inclusion of any header to enable the functionality in the Single UNIX Specification. Its definition subsumes the use of _POSIX_SOURCE and _POSIX_C_SOURCE.

An extract of code from a conforming application, that appears before any **#include** statements, is given below:

```
#define _XOPEN_SOURCE 600 /* Single UNIX Specification, Version 3 */
```

```
#include ...
```

Note that the definition of _XOPEN_SOURCE with the value 600 makes the definition of _POSIX_C_SOURCE redundant and it can safely be omitted.

B.2.2.2 The Name Space

The reservation of identifiers is paraphrased from the ISO C standard. The text is included because it needs to be part of IEEE Std 1003.1-200x, regardless of possible changes in future versions of the ISO C standard.

These identifiers may be used by implementations, particularly for feature test macros. Implementations should not use feature test macro names that might be reasonably used by a standard.

Including headers more than once is a reasonably common practice, and it should be carried forward from the ISO C standard. More significantly, having definitions in more than one header is explicitly permitted. Where the potential declaration is ‘‘benign’’ (the same definition twice) the declaration can be repeated, if that is permitted by the compiler. (This is usually true of macros, for example.) In those situations where a repetition is not benign (for example, **typedef**s), conditional compilation must be used. The situation actually occurs both within the ISO C standard and within POSIX.1: **time_t** should be in **<sys/types.h>**, and the ISO C standard mandates that it be in **<time.h>**.

The area of name space pollution *versus* additions to structures is difficult because of the macro structure of C. The following discussion summarizes all the various problems with and objections to the issue.

Note the phrase ‘‘user-defined macro’’. Users are not permitted to define macro names (or any other name) beginning with `"_[A-Z_]"`. Thus, the conflict cannot occur for symbols reserved to the vendor’s name space, and the permission to add fields automatically applies, without qualification, to those symbols.

1. Data structures (and unions) need to be defined in headers by implementations to meet certain requirements of POSIX.1 and the ISO C standard.

2. The structures defined by POSIX.1 are typically minimal, and any practical implementation would wish to add fields to these structures either to hold additional related information or for backwards-compatibility (or both). Future standards (and *de facto* standards) would also wish to add to these structures. Issues of field alignment make it impractical (at least in the general case) to simply omit fields when they are not defined by the particular standard involved.

   Struct **dirent** is an example of such a minimal structure (although one could argue about whether the other fields need visible names). The *st_rdev* field of most implementations’ **stat** structure is a common example where extension is needed and where a conflict could occur.

3. Fields in structures are in an independent name space, so the addition of such fields presents no problem to the C language itself in that such names cannot interact with identically named user symbols because access is qualified by the specific structure name.

4. There is an exception to this: macro processing is done at a lexical level. Thus, symbols added to a structure might be recognized as user-provided macro names at the location where the structure is declared. This only can occur if the user-provided name is declared as a macro before the header declaring the structure is included. The user’s use of the name after the declaration cannot interfere with the structure because the symbol is hidden and only accessible through access to the structure. Presumably, the user would not declare such a macro if there was an intention to use that field name.

5. Macros from the same or a related header might use the additional fields in the structure, and those field names might also collide with user macros. Although this is a less frequent occurrence, since macros are expanded at the point of use, no constraint on the order of use

of names can apply.

6. An ‘‘obvious’’ solution of using names in the reserved name space and then redefining them as macros when they should be visible does not work because this has the effect of exporting the symbol into the general name space. For example, given a (hypothetical) system-provided header **<h.h>**, and two parts of a C program in **a.c** and **b.c**, in header **<h.h>**:

   ```
   struct foo {
       int __i;
   }

   #ifdef _FEATURE_TEST
   #define i __i;
   #endif
   ```

   In file **a.c**:

   ```
   #include h.h
   extern int i;
   ...
   ```

   In file **b.c**:

   ```
   extern int i;
   ...
   ```

   The symbol that the user thinks of as *i* in both files has an external name of *__i* in **a.c**; the same symbol *i* in **b.c** has an external name *i* (ignoring any hidden manipulations the compiler might perform on the names). This would cause a mysterious name resolution problem when **a.o** and **b.o** are linked.

   Simply avoiding definition then causes alignment problems in the structure.

   A structure of the form:

   ```
   struct foo {
       union {
           int __i;
   #ifdef _FEATURE_TEST
           int i;
   #endif
       } __ii;
   }
   ```

   does not work because the name of the logical field *i* is *__ii.i*, and introduction of a macro to restore the logical name immediately reintroduces the problem discussed previously (although its manifestation might be more immediate because a syntax error would result if a recursive macro did not cause it to fail first).

7. A more workable solution would be to declare the structure:

```
struct foo {
#ifdef _FEATURE_TEST
    int i;
#else
    int __i;
#endif
}
```

However, if a macro (particularly one required by a standard) is to be defined that uses this field, two must be defined: one that uses *i*, the other that uses *__i*. If more than one additional field is used in a macro and they are conditional on distinct combinations of features, the complexity goes up as 2^n.

All this leaves a difficult situation: vendors must provide very complex headers to deal with what is conceptually simple and safe—adding a field to a structure. It is the possibility of user-provided macros with the same name that makes this difficult.

Several alternatives were proposed that involved constraining the user's access to part of the name space available to the user (as specified by the ISO C standard). In some cases, this was only until all the headers had been included. There were two proposals discussed that failed to achieve consensus:

1. Limiting it for the whole program.
2. Restricting the use of identifiers containing only uppercase letters until after all system headers had been included. It was also pointed out that because macros might wish to access fields of a structure (and macro expansion occurs totally at point of use) restricting names in this way would not protect the macro expansion, and thus the solution was inadequate.

It was finally decided that reservation of symbols would occur, but as constrained.

The current wording also allows the addition of fields to a structure, but requires that user macros of the same name not interfere. This allows vendors to do one of the following:

- Not create the situation (do not extend the structures with user-accessible names or use the solution in (7) above)
- Extend their compilers to allow some way of adding names to structures and macros safely

There are at least two ways that the compiler might be extended: add new preprocessor directives that turn off and on macro expansion for certain symbols (without changing the value of the macro) and a function or lexical operation that suppresses expansion of a word. The latter seems more flexible, particularly because it addresses the problem in macros as well as in declarations.

The following seems to be a possible implementation extension to the C language that will do this: any token that during macro expansion is found to be preceded by three `'#'` symbols shall not be further expanded in exactly the same way as described for macros that expand to their own name as in Section 3.8.3.4 of the ISO C standard. A vendor may also wish to implement this as an operation that is lexically a function, which might be implemented as:

```
#define __safe_name(x) ###x
```

Using a function notation would insulate vendors from changes in standards until such a functionality is standardized (if ever). Standardization of such a function would be valuable because it would then permit third parties to take advantage of it portably in software they may supply.

The symbols that are ‘‘explicitly permitted, but not required by IEEE Std 1003.1-200x’’ include those classified below. (That is, the symbols classified below might, but are not required to, be present when _POSIX_C_SOURCE is defined to have the value 20010xL.)

- Symbols in **<limits.h>** and **<unistd.h>** that are defined to indicate support for options or limits that are constant at compile-time.
- Symbols in the name space reserved for the implementation by the ISO C standard.
- Symbols in a name space reserved for a particular type of extension (for example, type names ending with **_t** in **<sys/types.h>**).
- Additional members of structures or unions whose names do not reduce the name space reserved for applications.

Since both implementations and future revisions of IEEE Std 1003.1-200x and other POSIX standards may use symbols in the reserved spaces described in these tables, there is a potential for name space clashes. To avoid future name space clashes when adding symbols, implementations should not use the posix_, POSIX_, or _POSIX_ prefixes.

B.2.3 Error Numbers

It was the consensus of the standard developers that to allow the conformance document to state that an error occurs and under what conditions, but to disallow a statement that it never occurs, does not make sense. It could be implied by the current wording that this is allowed, but to reduce the possibility of future interpretation requests, it is better to make an explicit statement.

The ISO C standard requires that *errno* be an assignable lvalue. Originally, the definition in POSIX.1 was stricter than that in the ISO C standard, **extern int** *errno*, in order to support historical usage. In a multi-threaded environment, implementing *errno* as a global variable results in non-deterministic results when accessed. It is required, however, that *errno* work as a per-thread error reporting mechanism. In order to do this, a separate *errno* value has to be maintained for each thread. The following section discusses the various alternative solutions that were considered. |

In order to avoid this problem altogether for new functions, these functions avoid using *errno* and, instead, return the error number directly as the function return value; a return value of zero indicates that no error was detected.

For any function that can return errors, the function return value is not used for any purpose other than for reporting errors. Even when the output of the function is scalar, it is passed through a function argument. While it might have been possible to allow some scalar outputs to be coded as negative function return values and mixed in with positive error status returns, this was rejected—using the return value for a mixed purpose was judged to be of limited use and error prone.

Checking the value of *errno* alone is not sufficient to determine the existence or type of an error, since it is not required that a successful function call clear *errno*. The variable *errno* should only be examined when the return value of a function indicates that the value of *errno* is meaningful. In that case, the function is required to set the variable to something other than zero.

The variable *errno* shall never be set to zero by any function call; to do so would contradict the ISO C standard.

POSIX.1 requires (in the ERRORS sections of function descriptions) certain error values to be set in certain conditions because many existing applications depend on them. Some error numbers, such as [EFAULT], are entirely implementation-defined and are noted as such in their description in the ERRORS section. This section otherwise allows wide latitude to the implementation in handling error reporting.

Some of the ERRORS sections in IEEE Std 1003.1-200x have two subsections. The first:

> ‘‘The function shall fail if:’’

could be called the ‘‘mandatory’’ section.

The second:

> ‘‘The function may fail if:’’

could be informally known as the ‘‘optional’’ section.

Attempting to infer the quality of an implementation based on whether it detects optional error conditions is not useful.

Following each one-word symbolic name for an error, there is a description of the error. The rationale for some of the symbolic names follows:

[ECANCELED]	This spelling was chosen as being more common.
[EFAULT]	Most historical implementations do not catch an error and set *errno* when an invalid address is given to the functions *wait*( ), *time*( ), or *times*( ). Some implementations cannot reliably detect an invalid address. And most systems that detect invalid addresses will do so only for a system call, not for a library routine.
[EFTYPE]	This error code was proposed in earlier proposals as ‘‘Inappropriate operation for file type’’, meaning that the operation requested is not appropriate for the file specified in the function call. This code was proposed, although the same idea was covered by [ENOTTY], because the connotations of the name would be misleading. It was pointed out that the *fcntl*( ) function uses the error code [EINVAL] for this notion, and hence all instances of [EFTYPE] were changed to this code.
[EINTR]	POSIX.1 prohibits conforming implementations from restarting interrupted system calls. However, it does not require that [EINTR] be returned when another legitimate value may be substituted; for example, a partial transfer count when *read*( ) or *write*( ) are interrupted. This is only given when the signal catching function returns normally as opposed to returns by mechanisms like *longjmp*( ) or *siglongjmp*( ).
[ELOOP]	In specifying conditions under which implementations would generate this error, the following goals were considered: • To ensure that actual loops are detected, including loops that result from symbolic links across distributed file systems. • To ensure that during pathname resolution an application can rely on the ability to follow at least {SYMLOOP_MAX} symbolic links in the absence of a loop. • To allow implementations to provide the capability of traversing more than {SYMLOOP_MAX} symbolic links in the absence of a loop. • To allow implementations to detect loops and generate the error prior to encountering {SYMLOOP_MAX} symbolic links.

[ENAMETOOLONG]	When a symbolic link is encountered during pathname resolution, the contents of that symbolic link are used to create a new pathname. The standard developers intended to allow, but not require, that implementations enforce the restriction of {PATH_MAX} on the result of this pathname substitution.
[ENOMEM]	The term *main memory* is not used in POSIX.1 because it is implementation-defined.
[ENOTSUP]	This error code is to be used when an implementation chooses to implement the required functionality of IEEE Std 1003.1-200x but does not support optional facilities defined by IEEE Std 1003.1-200x. The return of [ENOSYS] is to be taken to indicate that the function of the interface is not supported at all; the function will always fail with this error code.
[ENOTTY]	The symbolic name for this error is derived from a time when device control was done by *ioctl*( ) and that operation was only permitted on a terminal interface. The term *TTY* is derived from *teletypewriter*, the devices to which this error originally applied.
[EOVERFLOW]	Most of the uses of this error code are related to large file support. Typically, these cases occur on systems which support multiple programming environments with different sizes for **off_t**, but they may also occur in connection with remote file systems. In addition, when different programming environments have different widths for types such as **int** and **uid_t**, several functions may encounter a condition where a value in a particular environment is too wide to be represented. In that case, this error should be raised. For example, suppose the currently running process has 64-bit **int**, and file descriptor 9223372036854775807 is open and does not have the close-on-*exec* flag set. If the process then uses *execl*( ) to *exec* a file compiled in a programming environment with 32-bit **int**, the call to *execl*( ) can fail with *errno* set to [EOVERFLOW]. A similar failure can occur with *execl*( ) if any of the user IDs or any of the group IDs to be assigned to the new process image are out of range for the executed file's programming environment. Note, however, that this condition cannot occur for functions that are explicitly described as always being successful, such as *getpid*( ).
[EPIPE]	This condition normally generates the signal SIGPIPE; the error is returned if the signal does not terminate the process.
[EROFS]	In historical implementations, attempting to *unlink*( ) or *rmdir*( ) a mount point would generate an [EBUSY] error. An implementation could be envisioned where such an operation could be performed without error. In this case, if *either* the directory entry or the actual data structures reside on a read-only file system, [EROFS] is the appropriate error to generate. (For example, changing the link count of a file on a read-only file system could not be done, as is required by *unlink*( ), and thus an error should be reported.)

Three error numbers, [EDOM], [EILSEQ], and [ERANGE], were added to this section primarily for consistency with the ISO C standard.

Alternative Solutions for Per-Thread errno

The usual implementation of *errno* as a single global variable does not work in a multi-threaded environment. In such an environment, a thread may make a POSIX.1 call and get a −1 error return, but before that thread can check the value of *errno*, another thread might have made a second POSIX.1 call that also set *errno*. This behavior is unacceptable in robust programs. There were a number of alternatives that were considered for handling the *errno* problem:

- Implement *errno* as a per-thread integer variable.
- Implement *errno* as a service that can access the per-thread error number.
- Change all POSIX.1 calls to accept an extra status argument and avoid setting *errno*.
- Change all POSIX.1 calls to raise a language exception.

The first option offers the highest level of compatibility with existing practice but requires special support in the linker, compiler, and/or virtual memory system to support the new concept of thread private variables. When compared with current practice, the third and fourth options are much cleaner, more efficient, and encourage a more robust programming style, but they require new versions of all of the POSIX.1 functions that might detect an error. The second option offers compatibility with existing code that uses the **<errno.h>** header to define the symbol *errno*. In this option, *errno* may be a macro defined:

```
#define errno  (*__errno())
extern int      *__errno();
```

This option may be implemented as a per-thread variable whereby an *errno* field is allocated in the user space object representing a thread, and whereby the function *__errno*( ) makes a system call to determine the location of its user space object and returns the address of the *errno* field of that object. Another implementation, one that avoids calling the kernel, involves allocating stacks in chunks. The stack allocator keeps a side table indexed by chunk number containing a pointer to the thread object that uses that chunk. The *__errno*( ) function then looks at the stack pointer, determines the chunk number, and uses that as an index into the chunk table to find its thread object and thus its private value of *errno*. On most architectures, this can be done in four to five instructions. Some compilers may wish to implement *__errno*( ) inline to improve performance.

Disallowing Return of the [EINTR] Error Code

Many blocking interfaces defined by IEEE Std 1003.1-200x may return [EINTR] if interrupted during their execution by a signal handler. Blocking interfaces introduced under the Threads option do not have this property. Instead, they require that the interface appear to be atomic with respect to interruption. In particular, clients of blocking interfaces need not handle any possible [EINTR] return as a special case since it will never occur. If it is necessary to restart operations or complete incomplete operations following the execution of a signal handler, this is handled by the implementation, rather than by the application.

Requiring applications to handle [EINTR] errors on blocking interfaces has been shown to be a frequent source of often unreproducible bugs, and it adds no compelling value to the available functionality. Thus, blocking interfaces introduced for use by multi-threaded programs do not use this paradigm. In particular, in none of the functions *flockfile*( ), *pthread_cond_timedwait*( ), *pthread_cond_wait*( ), *pthread_join*( ), *pthread_mutex_lock*( ), and *sigwait*( ) did providing [EINTR] returns add value, or even particularly make sense. Thus, these functions do not provide for an [EINTR] return, even when interrupted by a signal handler. The same arguments can be applied to *sem_wait*( ), *sem_trywait*( ), *sigwaitinfo*( ), and *sigtimedwait*( ), but implementations are permitted to return [EINTR] error codes for these functions for compatibility with earlier versions of IEEE Std 1003.1-200x. Applications cannot rely on calls to these functions returning [EINTR] error codes when signals are delivered to the calling thread, but they should allow for the possibility.

B.2.3.1 Additional Error Numbers

The ISO C standard defines the name space for implementations to add additional error numbers.

B.2.4 Signal Concepts

Historical implementations of signals, using the *signal*( ) function, have shortcomings that make them unreliable for many application uses. Because of this, a new signal mechanism, based very closely on the one of 4.2 BSD and 4.3 BSD, was added to POSIX.1.

Signal Names

The restriction on the actual type used for **sigset_t** is intended to guarantee that these objects can always be assigned, have their address taken, and be passed as parameters by value. It is not intended that this type be a structure including pointers to other data structures, as that could impact the portability of applications performing such operations. A reasonable implementation could be a structure containing an array of some integer type.

The signals described in IEEE Std 1003.1-200x must have unique values so that they may be named as parameters of **case** statements in the body of a C language **switch** clause. However, implementation-defined signals may have values that overlap with each other or with signals specified in IEEE Std 1003.1-200x. An example of this is SIGABRT, which traditionally overlaps some other signal, such as SIGIOT.

SIGKILL, SIGTERM, SIGUSR1, and SIGUSR2 are ordinarily generated only through the explicit use of the *kill*( ) function, although some implementations generate SIGKILL under extraordinary circumstances. SIGTERM is traditionally the default signal sent by the *kill* command.

The signals SIGBUS, SIGEMT, SIGIOT, SIGTRAP, and SIGSYS were omitted from POSIX.1 because their behavior is implementation-defined and could not be adequately categorized. Conforming implementations may deliver these signals, but must document the circumstances under which they are delivered and note any restrictions concerning their delivery. The signals SIGFPE, SIGILL, and SIGSEGV are similar in that they also generally result only from programming errors. They were included in POSIX.1 because they do indicate three relatively well-categorized conditions. They are all defined by the ISO C standard and thus would have to be defined by any system with a ISO C standard binding, even if not explicitly included in POSIX.1.

There is very little that a Conforming POSIX.1 Application can do by catching, ignoring, or masking any of the signals SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGBUS, SIGSEGV, SIGSYS, or SIGFPE. They will generally be generated by the system only in cases of programming errors. While it may be desirable for some robust code (for example, a library routine) to be able to detect and recover from programming errors in other code, these signals are not nearly sufficient for that purpose. One portable use that does exist for these signals is that a command interpreter can recognize them as the cause of a process' termination (with *wait*( )) and print an appropriate message. The mnemonic tags for these signals are derived from their PDP-11 origin.

The signals SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU, and SIGCONT are provided for job control and are unchanged from 4.2 BSD. The signal SIGCHLD is also typically used by job control shells to detect children that have terminated or, as in 4.2 BSD, stopped.

Some implementations, including System V, have a signal named SIGCLD, which is similar to SIGCHLD in 4.2 BSD. POSIX.1 permits implementations to have a single signal with both names. POSIX.1 carefully specifies ways in which conforming applications can avoid the semantic differences between the two different implementations. The name SIGCHLD was chosen for POSIX.1 because most current application usages of it can remain unchanged in conforming applications. SIGCLD in System V has more cases of semantics that POSIX.1 does not specify, and thus applications using it are more likely to require changes in addition to the name change.

The signals SIGUSR1 and SIGUSR2 are commonly used by applications for notification of exceptional behavior and are described as ‘‘reserved as application-defined’’ so that such use is not prohibited. Implementations should not generate SIGUSR1 or SIGUSR2, except when explicitly requested by *kill*( ). It is recommended that libraries not use these two signals, as such use in libraries could interfere with their use by applications calling the libraries. If such use is unavoidable, it should be documented. It is prudent for non-portable libraries to use non-standard signals to avoid conflicts with use of standard signals by portable libraries.

There is no portable way for an application to catch or ignore non-standard signals. Some implementations define the range of signal numbers, so applications can install signal-catching functions for all of them. Unfortunately, implementation-defined signals often cause problems when caught or ignored by applications that do not understand the reason for the signal. While the desire exists for an application to be more robust by handling all possible signals (even those only generated by *kill*( )), no existing mechanism was found to be sufficiently portable to include in POSIX.1. The value of such a mechanism, if included, would be diminished given that SIGKILL would still not be catchable.

A number of new signal numbers are reserved for applications because the two user signals defined by POSIX.1 are insufficient for many realtime applications. A range of signal numbers is specified, rather than an enumeration of additional reserved signal names, because different applications and application profiles will require a different number of application signals. It is not desirable to burden all application domains and therefore all implementations with the maximum number of signals required by all possible applications. Note that in this context, signal numbers are essentially different signal priorities.

The relatively small number of required additional signals, {_POSIX_RTSIG_MAX}, was chosen so as not to require an unreasonably large signal mask/set. While this number of signals defined in POSIX.1 will fit in a single 32-bit word signal mask, it is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the ‘‘null signal’’). Support of {_POSIX_RTSIG_MAX} additional signals may push some implementation over the single 32-bit word line, but is unlikely to push any implementations that are already over that line beyond the 64-signal line.

B.2.4.1 Signal Generation and Delivery

The terms defined in this section are not used consistently in documentation of historical systems. Each signal can be considered to have a lifetime beginning with *generation* and ending with *delivery* or *acceptance*. The POSIX.1 definition of *delivery* does not exclude ignored signals; this is considered a more consistent definition. This revised text in several parts of IEEE Std 1003.1-200x clarifies the distinct semantics of asynchronous signal *delivery* and synchronous signal *acceptance*. The previous wording attempted to categorize both under the term *delivery*, which led to conflicts over whether the effects of asynchronous signal delivery applied to synchronous signal acceptance.

Signals generated for a process are delivered to only one thread. Thus, if more than one thread is eligible to receive a signal, one has to be chosen. The choice of threads is left entirely up to the implementation both to allow the widest possible range of conforming implementations and to give implementations the freedom to deliver the signal to the ‘‘easiest possible’’ thread should there be differences in ease of delivery between different threads.

Note that should multiple delivery among cooperating threads be required by an application, this can be trivially constructed out of the provided single-delivery semantics. The construction of a *sigwait_multiple*( ) function that accomplishes this goal is presented with the rationale for *sigwaitinfo*( ).

Implementations should deliver unblocked signals as soon after they are generated as possible. However, it is difficult for POSIX.1 to make specific requirements about this, beyond those in *kill*( ) and *sigprocmask*( ). Even on systems with prompt delivery, scheduling of higher priority processes is always likely to cause delays.

In general, the interval between the generation and delivery of unblocked signals cannot be detected by an application. Thus, references to pending signals generally apply to blocked, pending signals. An implementation registers a signal as pending on the process when no thread has the signal unblocked and there are no threads blocked in a *sigwait*( ) function for that signal. Thereafter, the implementation delivers the signal to the first thread that unblocks the signal or calls a *sigwait*( ) function on a signal set containing this signal rather than choosing the recipient thread at the time the signal is sent.

In the 4.3 BSD system, signals that are blocked and set to SIG_IGN are discarded immediately upon generation. For a signal that is ignored as its default action, if the action is SIG_DFL and the signal is blocked, a generated signal remains pending. In the 4.1 BSD system and in System V, Release 3, two other implementations that support a somewhat similar signal mechanism, all ignored, blocked signals remain pending if generated. Because it is not normally useful for an application to simultaneously ignore and block the same signal, it was unnecessary for POSIX.1 to specify behavior that would invalidate any of the historical implementations.

There is one case in some historical implementations where an unblocked, pending signal does not remain pending until it is delivered. In the System V implementation of *signal*( ), pending signals are discarded when the action is set to SIG_DFL or a signal-catching routine (as well as to SIG_IGN). Except in the case of setting SIGCHLD to SIG_DFL, implementations that do this do not conform completely to POSIX.1. Some earlier proposals for POSIX.1 explicitly stated this, but these statements were redundant due to the requirement that functions defined by POSIX.1 not change attributes of processes defined by POSIX.1 except as explicitly stated.

POSIX.1 specifically states that the order in which multiple, simultaneously pending signals are delivered is unspecified. This order has not been explicitly specified in historical implementations, but has remained quite consistent and been known to those familiar with the implementations. Thus, there have been cases where applications (usually system utilities) have been written with explicit or implicit dependencies on this order. Implementors and others porting existing applications may need to be aware of such dependencies.

When there are multiple pending signals that are not blocked, implementations should arrange for the delivery of all signals at once, if possible. Some implementations stack calls to all pending signal-catching routines, making it appear that each signal-catcher was interrupted by the next signal. In this case, the implementation should ensure that this stacking of signals does not violate the semantics of the signal masks established by *sigaction*( ). Other implementations process at most one signal when the operating system is entered, with remaining signals saved for later delivery. Although this practice is widespread, this behavior is neither standardized nor endorsed. In either case, implementations should attempt to deliver signals associated with the current state of the process (for example, SIGFPE) before other signals, if possible.

In 4.2 BSD and 4.3 BSD, it is not permissible to ignore or explicitly block SIGCONT, because if blocking or ignoring this signal prevented it from continuing a stopped process, such a process could never be continued (only killed by SIGKILL). However, 4.2 BSD and 4.3 BSD do block SIGCONT during execution of its signal-catching function when it is caught, creating exactly this problem. A proposal was considered to disallow catching SIGCONT in addition to ignoring and blocking it, but this limitation led to objections. The consensus was to require that SIGCONT always continue a stopped process when generated. This removed the need to disallow ignoring or explicit blocking of the signal; note that SIG_IGN and SIG_DFL are equivalent for SIGCONT .

B.2.4.2 Realtime Signal Generation and Delivery

The Realtime Signals Extension option to POSIX.1 signal generation and delivery behavior is required for the following reasons:

- The **sigevent** structure is used by other POSIX.1 functions that result in asynchronous event notifications to specify the notification mechanism to use and other information needed by the notification mechanism. IEEE Std 1003.1-200x defines only three symbolic values for the notification mechanism. SIGEV_NONE is used to indicate that no notification is required when the event occurs. This is useful for applications that use asynchronous I/O with polling for completion. SIGEV_SIGNAL indicates that a signal shall be generated when the event occurs. SIGEV_NOTIFY provides for ‘‘callback functions’’ for asynchronous notifications done by a function call within the context of a new thread. This provides a multi-threaded process a more natural means of notification than signals. The primary difficulty with previous notification approaches has been to specify the environment of the notification routine.

  — One approach is to limit the notification routine to call only functions permitted in a signal handler. While the list of permissible functions is clearly stated, this is overly restrictive.

  — A second approach is to define a new list of functions or classes of functions that are explicitly permitted or not permitted. This would give a programmer more lists to deal with, which would be awkward.

  — The third approach is to define completely the environment for execution of the notification function. A clear definition of an execution environment for notification is provided by executing the notification function in the environment of a newly created thread.

  Implementations may support additional notification mechanisms by defining new values for *sigev_notify* .

  For a notification type of SIGEV_SIGNAL, the other members of the **sigevent** structure defined by IEEE Std 1003.1-200x specify the realtime signal—that is, the signal number and application-defined value that differentiates between occurrences of signals with the same number—that will be generated when the event occurs. The structure is defined in **<signal.h>**, even though the structure is not directly used by any of the signal functions, because it is part of the signals interface used by the POSIX.1b ‘‘client functions’’. When the client functions include **<signal.h>** to define the signal names, the **sigevent** structure will also be defined.

  An application-defined value passed to the signal handler is used to differentiate between different ‘‘events’’ instead of requiring that the application use different signal numbers for several reasons:

  - — Realtime applications potentially handle a very large number of different events. Requiring that implementations support a correspondingly large number of distinct signal numbers will adversely impact the performance of signal delivery because the signal masks to be manipulated on entry and exit to the handlers will become large.
  - — Event notifications are prioritized by signal number (the rationale for this is explained in the following paragraphs) and the use of different signal numbers to differentiate between the different event notifications overloads the signal number more than has already been done. It also requires that the application writer make arbitrary assignments of priority to events that are logically of equal priority.

  A union is defined for the application-defined value so that either an integer constant or a pointer can be portably passed to the signal-catching function. On some architectures a pointer cannot be cast to an **int** and *vice versa*.

  Use of a structure here with an explicit notification type discriminant rather than explicit parameters to realtime functions, or embedded in other realtime structures, provides for future extensions to IEEE Std 1003.1-200x. Additional, perhaps more efficient, notification mechanisms can be supported for existing realtime function interfaces, such as timers and asynchronous I/O, by extending the **sigevent** structure appropriately. The existing realtime function interfaces will not have to be modified to use any such new notification mechanism. The revised text concerning the SIGEV_SIGNAL value makes consistent the semantics of the members of the **sigevent** structure, particularly in the definitions of *lio_listio*( ) and *aio_fsync*( ). For uniformity, other revisions cause this specification to be referred to rather than inaccurately duplicated in the descriptions of functions and structures using the **sigevent** structure. The revised wording does not relax the requirement that the signal number be in the range SIGRTMIN to SIGRTMAX to guarantee queuing and passing of the application value, since that requirement is still implied by the signal names.

- IEEE Std 1003.1-200x is intentionally vague on whether ''non-realtime'' signal-generating mechanisms can result in a **siginfo_t** being supplied to the handler on delivery. In one existing implementation, a **siginfo_t** is posted on signal generation, even though the implementation does not support queuing of multiple occurrences of a signal. It is not the intent of IEEE Std 1003.1-200x to preclude this, independent of the mandate to define signals that do support queuing. Any interpretation that appears to preclude this is a mistake in the reading or writing of the standard.

- Signals handled by realtime signal handlers might be generated by functions or conditions that do not allow the specification of an application-defined value and do not queue. IEEE Std 1003.1-200x specifies the *si_code* member of the **siginfo_t** structure used in existing practice and defines additional codes so that applications can detect whether an application-defined value is present or not. The code SI_USER for *kill*( )-generated signals is adopted from existing practice.

- The *sigaction*( ) *sa_flags* value SA_SIGINFO tells the implementation that the signal-catching function expects two additional arguments. When the flag is not set, a single argument, the signal number, is passed as specified by IEEE Std 1003.1-200x. Although IEEE Std 1003.1-200x does not explicitly allow the *info* argument to the handler function to be NULL, this is existing practice. This provides for compatibility with programs whose signal-catching functions are not prepared to accept the additional arguments. IEEE Std 1003.1-200x is explicitly unspecified as to whether signals actually queue when SA_SIGINFO is not set for a signal, as there appear to be no benefits to applications in specifying one behavior or another. One existing implementation queues a **siginfo_t** on each signal generation, unless the signal is already pending, in which case the implementation discards the new **siginfo_t**; that is, the queue length is never greater than one. This implementation only examines SA_SIGINFO on signal delivery, discarding the queued **siginfo_t** if its delivery was not requested.

IEEE Std 1003.1-200x specifies several new values for the *si_code* member of the **siginfo_t** structure. In existing practice, a *si_code* value of less than or equal to zero indicates that the signal was generated by a process via the *kill*( ) function. In existing practice, values of *si_code* that provide additional information for implementation-generated signals, such as SIGFPE or SIGSEGV , are all positive. Thus, if implementations define the new constants specified in IEEE Std 1003.1-200x to be negative numbers, programs written to use existing practice will not break. IEEE Std 1003.1-200x chose not to attempt to specify existing practice values of *si_code* other than SI_USER both because it was deemed beyond the scope of IEEE Std 1003.1-200x and because many of the values in existing practice appear to be platform and implementation-defined. But, IEEE Std 1003.1-200x does specify that if an implementation—for example, one that does not have existing practice in this area—chooses to define additional values for *si_code*, these values have to be different from the values of the symbols specified by IEEE Std 1003.1-200x. This will allow conforming applications to differentiate between signals generated by one of the POSIX.1b asynchronous events and those generated by other implementation events in a manner compatible with existing practice.

The unique values of *si_code* for the POSIX.1b asynchronous events have implications for implementations of, for example, asynchronous I/O or message passing in user space library code. Such an implementation will be required to provide a hidden interface to the signal generation mechanism that allows the library to specify the standard values of *si_code*.

Existing practice also defines additional members of **siginfo_t**, such as the process ID and user ID of the sending process for *kill*( )-generated signals. These members were deemed not necessary to meet the requirements of realtime applications and are not specified by IEEE Std 1003.1-200x. Neither are they precluded.

The third argument to the signal-catching function, *context*, is left undefined by IEEE Std 1003.1-200x, but is specified in the interface because it matches existing practice for the SA_SIGINFO flag. It was considered undesirable to require a separate implementation for SA_SIGINFO for POSIX conformance on implementations that already support the two additional parameters.

- The requirement to deliver lower numbered signals in the range SIGRTMIN to SIGRTMAX first, when multiple unblocked signals are pending, results from several considerations:

  — A method is required to prioritize event notifications. The signal number was chosen instead of, for instance, associating a separate priority with each request, because an implementation has to check pending signals at various points and select one for delivery when more than one is pending. Specifying a selection order is the minimal additional semantic that will achieve prioritized delivery. If a separate priority were to be associated with queued signals, it would be necessary for an implementation to search all non-empty, non-blocked signal queues and select from among them the pending signal with the highest priority. This would significantly increase the cost of and decrease the determinism of signal delivery.

  — Given the specified selection of the lowest numeric unblocked pending signal, preemptive priority signal delivery can be achieved using signal numbers and signal masks by ensuring that the *sa_mask* for each signal number blocks all signals with a higher numeric value.

  For realtime applications that want to use only the newly defined realtime signal numbers without interference from the standard signals, this can be achieved by blocking all of the standard signals in the process signal mask and in the *sa_mask* installed by the signal action for the realtime signal handlers.

IEEE Std 1003.1-200x explicitly leaves unspecified the ordering of signals outside of the range of realtime signals and the ordering of signals within this range with respect to those outside the range. It was believed that this would unduly constrain implementations or standards in the future definition of new signals.

B.2.4.3 Signal Actions

Early proposals mentioned SIGCONT as a second exception to the rule that signals are not delivered to stopped processes until continued. Because IEEE Std 1003.1-200x now specifies that SIGCONT causes the stopped process to continue when it is generated, delivery of SIGCONT is not prevented because a process is stopped, even without an explicit exception to this rule.

Ignoring a signal by setting the action to SIG_IGN (or SIG_DFL for signals whose default action is to ignore) is not the same as installing a signal-catching function that simply returns. Invoking such a function will interrupt certain system functions that block processes (for example, *wait*( ), *sigsuspend*( ), *pause*( ), *read*( ), *write*( )) while ignoring a signal has no such effect on the process.

Historical implementations discard pending signals when the action is set to SIG_IGN. However, they do not always do the same when the action is set to SIG_DFL and the default action is to ignore the signal. IEEE Std 1003.1-200x requires this for the sake of consistency and also for completeness, since the only signal this applies to is SIGCHLD, and IEEE Std 1003.1-200x disallows setting its action to SIG_IGN.

Some implementations (System V, for example) assign different semantics for SIGCLD depending on whether the action is set to SIG_IGN or SIG_DFL. Since POSIX.1 requires that the default action for SIGCHLD be to ignore the signal, applications should always set the action to SIG_DFL in order to avoid SIGCHLD.

Whether or not an implementation allows SIG_IGN as a SIGCHLD disposition to be inherited across a call to one of the *exec* family of functions or *posix_spawn*( ) is explicitly left as unspecified. This change was made as a result of IEEE PASC Interpretation 1003.1 #132, and permits the implementation to decide between the following alternatives:

- Unconditionally leave SIGCHLD set to SIG_IGN, in which case the implementation would not allow applications that assume inheritance of SIG_DFL to conform to IEEE Std 1003.1-200x without change. The implementation would, however, retain an ability to control applications that create child processes but never call on the *wait* family of functions, potentially filling up the process table.
- Unconditionally reset SIGCHLD to SIG_DFL, in which case the implementation would allow applications that assume inheritance of SIG_DFL to conform. The implementation would, however, lose an ability to control applications that spawn child processes but never reap them.
- Provide some mechanism, not specified in IEEE Std 1003.1-200x, to control inherited SIGCHLD dispositions.

Some implementations (System V, for example) will deliver a SIGCLD signal immediately when a process establishes a signal-catching function for SIGCLD when that process has a child that has already terminated. Other implementations, such as 4.3 BSD, do not generate a new SIGCHLD signal in this way. In general, a process should not attempt to alter the signal action for the SIGCHLD signal while it has any outstanding children. However, it is not always possible for a process to avoid this; for example, shells sometimes start up processes in pipelines with other processes from the pipeline as children. Processes that cannot ensure that they have no children when altering the signal action for SIGCHLD thus need to be prepared for, but not depend on, generation of an immediate SIGCHLD signal.

The default action of the stop signals (SIGSTOP , SIGTSTP, SIGTTIN, SIGTTOU) is to stop a process that is executing. If a stop signal is delivered to a process that is already stopped, it has no effect. In fact, if a stop signal is generated for a stopped process whose signal mask blocks the signal, the signal will never be delivered to the process since the process must receive a SIGCONT, which discards all pending stop signals, in order to continue executing.

The SIGCONT signal shall continue a stopped process even if SIGCONT is blocked (or ignored). However, if a signal-catching routine has been established for SIGCONT, it will not be entered until SIGCONT is unblocked.

If a process in an orphaned process group stops, it is no longer under the control of a job control shell and hence would not normally ever be continued. Because of this, orphaned processes that receive terminal-related stop signals (SIGTSTP , SIGTTIN, SIGTTOU, but not SIGSTOP) must not be allowed to stop. The goal is to prevent stopped processes from languishing forever. (As SIGSTOP is sent only via *kill*( ), it is assumed that the process or user sending a SIGSTOP can send a SIGCONT when desired.) Instead, the system must discard the stop signal. As an extension, it may also deliver another signal in its place. 4.3 BSD sends a SIGKILL, which is overly effective because SIGKILL is not catchable. Another possible choice is SIGHUP. 4.3 BSD also does this for orphaned processes (processes whose parent has terminated) rather than for members of orphaned process groups; this is less desirable because job control shells manage process groups. POSIX.1 also prevents SIGTTIN and SIGTTOU signals from being generated for processes in orphaned process groups as a direct result of activity on a terminal, preventing infinite loops when *read*( ) and *write*( ) calls generate signals that are discarded; see Section A.11.1.4 (on page 3357). A similar restriction on the generation of SIGTSTP was considered, but that would be unnecessary and more difficult to implement due to its asynchronous nature.

Although POSIX.1 requires that signal-catching functions be called with only one argument, there is nothing to prevent conforming implementations from extending POSIX.1 to pass additional arguments, as long as Strictly Conforming POSIX.1 Applications continue to compile and execute correctly. Most historical implementations do, in fact, pass additional, signal-specific arguments to certain signal-catching routines.

There was a proposal to change the declared type of the signal handler to:

```
void func (int sig, ...);
```

The usage of ellipses (`"..."`) is ISO C standard syntax to indicate a variable number of arguments. Its use was intended to allow the implementation to pass additional information to the signal handler in a standard manner.

Unfortunately, this construct would require all signal handlers to be defined with this syntax because the ISO C standard allows implementations to use a different parameter passing mechanism for variable parameter lists than for non-variable parameter lists. Thus, all existing signal handlers in all existing applications would have to be changed to use the variable syntax in order to be standard and portable. This is in conflict with the goal of Minimal Changes to Existing Application Code.

When terminating a process from a signal-catching function, processes should be aware of any interpretation that their parent may make of the status returned by *wait*( ) or *waitpid*( ). In particular, a signal-catching function should not call *exit*(0) or *_exit*(0) unless it wants to indicate successful termination. A non-zero argument to *exit*( ) or *_exit*( ) can be used to indicate unsuccessful termination. Alternatively, the process can use *kill*( ) to send itself a fatal signal (first ensuring that the signal is set to the default action and not blocked). See also the RATIONALE section of the *_exit*( ) function.

The behavior of *unsafe* functions, as defined by this section, is undefined when they are invoked from signal-catching functions in certain circumstances. The behavior of reentrant functions, as defined by this section, is as specified by POSIX.1, regardless of invocation from a signal-catching function. This is the only intended meaning of the statement that reentrant functions may be used in signal-catching functions without restriction. Applications must still consider all effects of such functions on such things as data structures, files, and process state. In particular, application writers need to consider the restrictions on interactions when interrupting *sleep*( ) (see *sleep*( )) and interactions among multiple handles for a file description. The fact that any specific function is listed as reentrant does not necessarily mean that invocation of that function from a signal-catching function is recommended.

In order to prevent errors arising from interrupting non-reentrant function calls, applications should protect calls to these functions either by blocking the appropriate signals or through the use of some programmatic semaphore. POSIX.1 does not address the more general problem of synchronizing access to shared data structures. Note in particular that even the ‘‘safe’’ functions may modify the global variable *errno*; the signal-catching function may want to save and restore its value. The same principles apply to the reentrancy of application routines and asynchronous data access.

Note that *longjmp*( ) and *siglongjmp*( ) are not in the list of reentrant functions. This is because the code executing after *longjmp*( ) or *siglongjmp*( ) can call any unsafe functions with the same danger as calling those unsafe functions directly from the signal handler. Applications that use *longjmp*( ) or *siglongjmp*( ) out of signal handlers require rigorous protection in order to be portable. Many of the other functions that are excluded from the list are traditionally implemented using either the C language *malloc*( ) or *free*( ) functions or the ISO C standard I/O library, both of which traditionally use data structures in a non-reentrant manner. Because any combination of different functions using a common data structure can cause reentrancy problems, POSIX.1 does not define the behavior when any unsafe function is called in a signal handler that interrupts any unsafe function.

The only realtime extension to signal actions is the addition of the additional parameters to the signal-catching function. This extension has been explained and motivated in the previous section. In making this extension, though, developers of POSIX.1b ran into issues relating to function prototypes. In response to input from the POSIX.1 standard developers, members were added to the **sigaction** structure to specify function prototypes for the newer signal-catching function specified by POSIX.1b. These members follow changes that are being made to POSIX.1. Note that IEEE Std 1003.1-200x explicitly states that these fields may overlap so that a union can be defined. This will enable existing implementations of POSIX.1 to maintain binary-compatibility when these extensions are added.

The **siginfo_t** structure was adopted for passing the application-defined value to match existing practice, but the existing practice has no provision for an application-defined value, so this was added. Note that POSIX normally reserves the ‘‘_t’’ type designation for opaque types. The **siginfo_t** structure breaks with this convention to follow existing practice and thus promote portability. Standardization of the existing practice for the other members of this structure may be addressed in the future.

Although it is not explicitly visible to applications, there are additional semantics for signal actions implied by queued signals and their interaction with other POSIX.1b realtime functions. Specifically:

- It is not necessary to queue signals whose action is SIG_IGN.
- For implementations that support POSIX.1b timers, some interaction with the timer functions at signal delivery is implied to manage the timer overrun count.

B.2.4.4 *Signal Effects on Other Functions*

The most common behavior of an interrupted function after a signal-catching function returns is for the interrupted function to give an [EINTR] error. However, there are a number of specific exceptions, including *sleep*( ) and certain situations with *read*( ) and *write*( ).

The historical implementations of many functions defined by IEEE Std 1003.1-200x are not interruptible, but delay delivery of signals generated during their execution until after they complete. This is never a problem for functions that are guaranteed to complete in a short (imperceptible to a human) period of time. It is normally those functions that can suspend a process indefinitely or for long periods of time (for example, *wait*( ), *pause*( ), *sigsuspend*( ), *sleep*( ), or *read*( )/*write*( ) on a slow device like a terminal] that are interruptible. This permits applications to respond to interactive signals or to set timeouts on calls to most such functions with *alarm*( ). Therefore, implementations should generally make such functions (including ones defined as extensions) interruptible.

Functions not mentioned explicitly as interruptible may be so on some implementations, possibly as an extension where the function gives an [EINTR] error. There are several functions (for example, *getpid*( ), *getuid*( )) that are specified as never returning an error, which can thus never be extended in this way.

B.2.5 Standard I/O Streams

B.2.5.1 *Interaction of File Descriptors and Standard I/O Streams*

There is no additional rationale provided for this section.

B.2.5.2 *Stream Orientation and Encoding Rules*

There is no additional rationale provided for this section.

B.2.6 STREAMS

STREAMS are introduced into IEEE Std 1003.1-200x as part of the alignment with the Single UNIX Specification, but marked as an option in recognition that not all systems may wish to implement the facility. The option within IEEE Std 1003.1-200x is denoted by the XSR margin marker. The standard developers made this option independent of the XSI option.

STREAMS are a method of implementing network services and other character-based input/output mechanisms, with the STREAM being a full-duplex connection between a process and a device. STREAMS provides direct access to protocol modules, and optional protocol modules can be interposed between the process-end of the STREAM and the device-driver at the device-end of the STREAM. Pipes can be implemented using the STREAMS mechanism, so they can provide process-to-process as well as process-to-device communications.

This section introduces STREAMS I/O, the message types used to control them, an overview of the priority mechanism, and the interfaces used to access them.

B.2.6.1 *Accessing STREAMS*

There is no additional rationale provided for this section.

B.2.7 XSI Interprocess Communication

There are two forms of IPC supported as options in IEEE Std 1003.1-200x. The traditional System V IPC routines derived from the SVID—that is, the *msg**( ), *sem**( ), and *shm**( ) interfaces—are mandatory on XSI-conformant systems. Thus, all XSI-conformant systems provide the same mechanisms for manipulating messages, shared memory, and semaphores.

In addition, the POSIX Realtime Extension provides an alternate set of routines for those systems supporting the appropriate options.

The application writer is presented with a choice: the System V interfaces or the POSIX interfaces (loosely derived from the Berkeley interfaces). The XSI profile prefers the System V interfaces, but the POSIX interfaces may be more suitable for realtime or other performance-sensitive applications.

B.2.7.1 IPC General Information

General information that is shared by all three mechanisms is described in this section. The common permissions mechanism is briefly introduced, describing the mode bits, and how they are used to determine whether or not a process has access to read or write/alter the appropriate instance of one of the IPC mechanisms. All other relevant information is contained in the reference pages themselves.

The semaphore type of IPC allows processes to communicate through the exchange of semaphore values. A semaphore is a positive integer. Since many applications require the use of more than one semaphore, XSI-conformant systems have the ability to create sets or arrays of semaphores.

Calls to support semaphores include:

semctl( ), *semget*( ), *semop*( )

Semaphore sets are created by using the *semget*( ) function.

The message type of IPC allows process to communicate through the exchange of data stored in buffers. This data is transmitted between processes in discrete portions known as messages.

Calls to support message queues include:

msgctl( ), *msgget*( ), *msgrcv*( ), *msgsnd*( )

The share memory type of IPC allows two or more processes to share memory and consequently the data contained therein. This is done by allowing processes to set up access to a common memory address space. This sharing of memory provides a fast means of exchange of data between processes.

Calls to support shared memory include:

shmctl( ), *shmdt*( ), *shmget*( )

The *ftok*( ) interface is also provided.

B.2.8 Realtime

Advisory Information

POSIX.1b contains an Informative Annex with proposed interfaces for ‘‘real-time files’’. These interfaces could determine groups of the exact parameters required to do ‘‘direct I/O’’ or ‘‘extents’’. These interfaces were objected to by a significant portion of the balloting group as too complex. A conforming application had little chance of correctly navigating the large parameter space to match its desires to the system. In addition, they only applied to a new type of file (realtime files) and they told the implementation exactly what to do as opposed to advising the implementation on application behavior and letting it optimize for the system the (portable) application was running on. For example, it was not clear how a system that had a disk array should set its parameters.

There seemed to be several overall goals:

- Optimizing sequential access
- Optimizing caching behavior
- Optimizing I/O data transfer
- Preallocation

The advisory interfaces, *posix_fadvise*( ) and *posix_madvise*( ), satisfy the first two goals. The POSIX_FADV_SEQUENTIAL and POSIX_MADV_SEQUENTIAL advice tells the implementation to expect serial access. Typically the system will prefetch the next several serial accesses in order to overlap I/O. It may also free previously accessed serial data if memory is tight. If the application is not doing serial access it can use POSIX_FADV_WILLNEED and POSIX_MADV_WILLNEED to accomplish I/O overlap, as required. When the application advises POSIX_FADV_RANDOM or POSIX_MADV_RANDOM behavior, the implementation usually tries to fetch a minimum amount of data with each request and it does not expect much locality. POSIX_FADV_DONTNEED and POSIX_MADV_DONTNEED allow the system to free up caching resources as the data will not be required in the near future.

POSIX_FADV_NOREUSE tells the system that caching the specified data is not optimal. For file I/O, the transfer should go directly to the user buffer instead of being cached internally by the implementation. To portably perform direct disk I/O on all systems, the application must perform its I/O transfers according to the following rules:

1. The user buffer should be aligned according to the {POSIX_REC_XFER_ALIGN} *pathconf*( ) variable.
2. The number of bytes transferred in an I/O operation should be a multiple of the {POSIX_ALLOC_SIZE_MIN} *pathconf*( ) variable.
3. The offset into the file at the start of an I/O operation should be a multiple of the {POSIX_ALLOC_SIZE_MIN} *pathconf*( ) variable.
4. The application should ensure that all threads which open a given file specify POSIX_FADV_NOREUSE to be sure that there is no unexpected interaction between threads using buffered I/O and threads using direct I/O to the same file.

In some cases, a user buffer must be properly aligned in order to be transferred directly to/from the device. The {POSIX_REC_XFER_ALIGN} *pathconf*( ) variable tells the application the proper alignment.

The preallocation goal is met by the space control function, *posix_fallocate*( ). The application can use *posix_fallocate*( ) to guarantee no [ENOSPC] errors and to improve performance by prepaying any overhead required for block allocation.

Implementations may use information conveyed by a previous *posix_fadvise*( ) call to influence the manner in which allocation is performed. For example, if an application did the following calls:

```
fd = open("file");
posix_fadvise(fd, offset, len, POSIX_FADV_SEQUENTIAL);
posix_fallocate(fd, len, size);
```

an implementation might allocate the file contiguously on disk.

Finally, the *pathconf*( ) variables {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE}, and {POSIX_REC_INCR_XFER_SIZE} tell the application a range of transfer sizes that are recommended for best I/O performance.

Where bounded response time is required, the vendor can supply the appropriate settings of the advisories to achieve a guaranteed performance level.

The interfaces meet the goals while allowing applications using regular files to take advantage of performance optimizations. The interfaces tell the implementation expected application behavior which the implementation can use to optimize performance on a particular system with a particular dynamic load.

The *posix_memalign*( ) function was added to allow for the allocation of specifically aligned buffers; for example, for {POSIX_REC_XFER_ALIGN}.

The working group also considered the alternative of adding a function which would return an aligned pointer to memory within a user supplied buffer. This was not considered to be the best method, because it potentially wastes large amounts of memory when buffers need to be aligned on large alignment boundaries.

Message Passing

This section provides the rationale for the definition of the message passing interface in IEEE Std 1003.1-200x. This is presented in terms of the objectives, models, and requirements imposed upon this interface.

- Objectives

  Many applications, including both realtime and database applications, require a means of passing arbitrary amounts of data between cooperating processes comprising the overall application on one or more processors. Many conventional interfaces for interprocess communication are insufficient for realtime applications in that efficient and deterministic data passing methods cannot be implemented. This has prompted the definition of message passing interfaces providing these facilities:

  — Open a message queue.

  — Send a message to a message queue.

  — Receive a message from a queue, either synchronously or asynchronously.

  — Alter message queue attributes for flow and resource control.

  It is assumed that an application may consist of multiple cooperating processes and that these processes may wish to communicate and coordinate their activities. The message passing facility described in IEEE Std 1003.1-200x allows processes to communicate through system-wide queues. These message queues are accessed through names that may be pathnames. A message queue can be opened for use by multiple sending and/or multiple receiving processes.

- Background on Embedded Applications

  Interprocess communication utilizing message passing is a key facility for the construction of deterministic, high-performance realtime applications. The facility is present in all realtime systems and is the framework upon which the application is constructed. The performance of the facility is usually a direct indication of the performance of the resulting application.

  Realtime applications, especially for embedded systems, are typically designed around the performance constraints imposed by the message passing mechanisms. Applications for embedded systems are typically very tightly constrained. Application writers expect to design and control the entire system. In order to minimize system costs, the writer will attempt to use all resources to their utmost and minimize the requirement to add additional memory or processors.

  The embedded applications usually share address spaces and only a simple message passing mechanism is required. The application can readily access common data incurring only mutual-exclusion overheads. The models desired are the simplest possible with the application building higher-level facilities only when needed.

- Requirements

  The following requirements determined the features of the message passing facilities defined in IEEE Std 1003.1-200x:

  — Naming of Message Queues

  The mechanism for gaining access to a message queue is a pathname evaluated in a context that is allowed to be a file system name space, or it can be independent of any file system. This is a specific attempt to allow implementations based on either method in order to address both embedded systems and to also allow implementation in larger systems.

  The interface of *mq_open*( ) is defined to allow but not require the access control and name conflicts resulting from utilizing a file system for name resolution. All required behavior is specified for the access control case. Yet a conforming implementation, such as an embedded system kernel, may define that there are no distinctions between users and may define that all process have all access privileges.

  — Embedded System Naming

  Embedded systems need to be able to utilize independent name spaces for accessing the various system objects. They typically do not have a file system, precluding its utilization as a common name resolution mechanism. The modularity of an embedded system limits the connections between separate mechanisms that can be allowed.

  Embedded systems typically do not have any access protection. Since the system does not support the mixing of applications from different areas, and usually does not even have the concept of an authorization entity, access control is not useful.

  — Large System Naming

  On systems with more functionality, the name resolution must support the ability to use the file system as the name resolution mechanism/object storage medium and to have control over access to the objects. Utilizing the pathname space can result in further errors when the names conflict with other objects.

  — Fixed Size of Messages

The interfaces impose a fixed upper bound on the size of messages that can be sent to a specific message queue. The size is set on an individual queue basis and cannot be changed dynamically.

The purpose of the fixed size is to increase the ability of the system to optimize the implementation of *mq_send*( ) and *mq_receive*( ). With fixed sizes of messages and fixed numbers of messages, specific message blocks can be pre-allocated. This eliminates a significant amount of checking for errors and boundary conditions. Additionally, an implementation can optimize data copying to maximize performance. Finally, with a restricted range of message sizes, an implementation is better able to provide deterministic operations.

— Prioritization of Messages

Message prioritization allows the application to determine the order in which messages are received. Prioritization of messages is a key facility that is provided by most realtime kernels and is heavily utilized by the applications. The major purpose of having priorities in message queues is to avoid priority inversions in the message system, where a high-priority message is delayed behind one or more lower-priority messages. This allows the applications to be designed so that they do not need to be interrupted in order to change the flow of control when exceptional conditions occur. The prioritization does add additional overhead to the message operations in those cases it is actually used but a clever implementation can optimize for the FIFO case to make that more efficient.

— Asynchronous Notification

The interface supports the ability to have a task asynchronously notified of the availability of a message on the queue. The purpose of this facility is to allow the task to perform other functions and yet still be notified that a message has become available on the queue.

To understand the requirement for this function, it is useful to understand two models of application design: a single task performing multiple functions and multiple tasks performing a single function. Each of these models has advantages.

Asynchronous notification is required to build the model of a single task performing multiple operations. This model typically results from either the expectation that interruption is less expensive than utilizing a separate task or from the growth of the application to include additional functions.

Semaphores

Semaphores are a high-performance process synchronization mechanism. Semaphores are named by null-terminated strings of characters.

A semaphore is created using the *sem_init*( ) function or the *sem_open*( ) function with the O_CREAT flag set in *oflag*.

To use a semaphore, a process has to first initialize the semaphore or inherit an open descriptor for the semaphore via *fork*( ).

A semaphore preserves its state when the last reference is closed. For example, if a semaphore has a value of 13 when the last reference is closed, it will have a value of 13 when it is next opened.

When a semaphore is created, an initial state for the semaphore has to be provided. This value is a non-negative integer. Negative values are not possible since they indicate the presence of blocked processes. The persistence of any of these objects across a system crash or a system reboot is undefined. Conforming applications shall not depend on any sort of persistence across a system reboot or a system crash.

- Models and Requirements

  A realtime system requires synchronization and communication between the processes comprising the overall application. An efficient and reliable synchronization mechanism has to be provided in a realtime system that will allow more than one schedulable process mutually-exclusive access to the same resource. This synchronization mechanism has to allow for the optimal implementation of synchronization or systems implementors will define other, more cost-effective methods.

  At issue are the methods whereby multiple processes (tasks) can be designed and implemented to work together in order to perform a single function. This requires interprocess communication and synchronization. A semaphore mechanism is the lowest level of synchronization that can be provided by an operating system.

  A semaphore is defined as an object that has an integral value and a set of blocked processes associated with it. If the value is positive or zero, then the set of blocked processes is empty; otherwise, the size of the set is equal to the absolute value of the semaphore value. The value of the semaphore can be incremented or decremented by any process with access to the semaphore and must be done as an indivisible operation. When a semaphore value is less than or equal to zero, any process that attempts to lock it again will block or be informed that it is not possible to perform the operation.

  A semaphore may be used to guard access to any resource accessible by more than one schedulable task in the system. It is a global entity and not associated with any particular process. As such, a method of obtaining access to the semaphore has to be provided by the operating system. A process that wants access to a critical resource (section) has to wait on the semaphore that guards that resource. When the semaphore is locked on behalf of a process, it knows that it can utilize the resource without interference by any other cooperating process in the system. When the process finishes its operation on the resource, leaving it in a well-defined state, it posts the semaphore, indicating that some other process may now obtain the resource associated with that semaphore.

  In this section, mutexes and condition variables are specified as the synchronization mechanisms between threads.

  These primitives are typically used for synchronizing threads that share memory in a single process. However, this section provides an option allowing the use of these synchronization interfaces and objects between processes that share memory, regardless of the method for sharing memory.

  Much experience with semaphores shows that there are two distinct uses of synchronization: locking, which is typically of short duration; and waiting, which is typically of long or unbounded duration. These distinct usages map directly onto mutexes and condition variables, respectively.

  Semaphores are provided in IEEE Std 1003.1-200x primarily to provide a means of synchronization for processes; these processes may or may not share memory. Mutexes and condition variables are specified as synchronization mechanisms between threads; these threads always share (some) memory. Both are synchronization paradigms that have been in widespread use for a number of years. Each set of primitives is particularly well matched to certain problems.

  With respect to binary semaphores, experience has shown that condition variables and mutexes are easier to use for many synchronization problems than binary semaphores. The primary reason for this is the explicit appearance of a Boolean predicate that specifies when the condition wait is satisfied. This Boolean predicate terminates a loop, including the call to *pthread_cond_wait*( ). As a result, extra wakeups are benign since the predicate governs whether the thread will actually proceed past the condition wait. With stateful primitives, such as binary semaphores, the wakeup in itself typically means that the wait is satisfied. The burden of ensuring correctness for such waits is thus placed on *all* signalers of the semaphore rather than on an *explicitly coded* Boolean predicate located at the condition wait. Experience has shown that the latter creates a major improvement in safety and ease-of-use.

Counting semaphores are well matched to dealing with producer/consumer problems, including those that might exist between threads of different processes, or between a signal handler and a thread. In the former case, there may be little or no memory shared by the processes; in the latter case, one is not communicating between co-equal threads, but between a thread and an interruptlike entity. It is for these reasons that IEEE Std 1003.1-200x allows semaphores to be used by threads.

Mutexes and condition variables have been effectively used with and without priority inheritance, priority ceiling, and other attributes to synchronize threads that share memory. The efficiency of their implementation is comparable to or better than that of other synchronization primitives that are sometimes harder to use (for example, binary semaphores). Furthermore, there is at least one known implementation of Ada tasking that uses these primitives. Mutexes and condition variables together constitute an appropriate, sufficient, and complete set of interthread synchronization primitives.

Efficient multi-threaded applications require high-performance synchronization primitives. Considerations of efficiency and generality require a small set of primitives upon which more sophisticated synchronization functions can be built.

- Standardization Issues

  It is possible to implement very high-performance semaphores using test-and-set instructions on shared memory locations. The library routines that implement such a high-performance interface has to properly ensure that a *sem_wait*( ) or *sem_trywait*( ) operation that cannot be performed will issue a blocking semaphore system call or properly report the condition to the application. The same interface to the application program would be provided by a high-performance implementation.

B.2.8.1 Realtime Signals

Realtime Signals Extension

This portion of the rationale presents models, requirements, and standardization issues relevant to the Realtime Signals Extension. This extension provides the capability required to support reliable, deterministic, asynchronous notification of events. While a new mechanism, unencumbered by the historical usage and semantics of POSIX.1 signals, might allow for a more efficient implementation, the application requirements for event notification can be met with a small number of extensions to signals. Therefore, a minimal set of extensions to signals to support the application requirements is specified.

The realtime signal extensions specified in this section are used by other realtime functions requiring asynchronous notification:

- Models

  The model supported is one of multiple cooperating processes, each of which handles multiple asynchronous external events. Events represent occurrences that are generated as the result of some activity in the system. Examples of occurrences that can constitute an event include:

— Completion of an asynchronous I/O request

— Expiration of a POSIX.1b timer

— Arrival of an interprocess message

— Generation of a user-defined event

Processing of these events may occur synchronously via polling for event notifications or asynchronously via a software interrupt mechanism. Existing practice for this model is well established for traditional proprietary realtime operating systems, realtime executives, and realtime extended POSIX-like systems.

A contrasting model is that of ‘‘cooperating sequential processes’’ where each process handles a single priority of events via polling. Each process blocks while waiting for events, and each process depends on the preemptive, priority-based process scheduling mechanism to arbitrate between events of different priority that need to be processed concurrently. Existing practice for this model is also well established for small realtime executives that typically execute in an unprotected physical address space, but it is just emerging in the context of a fuller function operating system with multiple virtual address spaces.

It could be argued that the cooperating sequential process model, and the facilities supported by the POSIX Threads Extension obviate a software interrupt model. But, even with the cooperating sequential process model, the need has been recognized for a software interrupt model to handle exceptional conditions and process aborting, so the mechanism must be supported in any case. Furthermore, it is not the purview of IEEE Std 1003.1-200x to attempt to convince realtime practitioners that their current application models based on software interrupts are ‘‘broken’’ and should be replaced by the cooperating sequential process model. Rather, it is the charter of IEEE Std 1003.1-200x to provide standard extensions to mechanisms that support existing realtime practice.

- Requirements

This section discusses the following realtime application requirements for asynchronous event notification:

— Reliable delivery of asynchronous event notification

The events notification mechanism shall guarantee delivery of an event notification. Asynchronous operations (such as asynchronous I/O and timers) that complete significantly after they are invoked have to guarantee that delivery of the event notification can occur at the time of completion.

— Prioritized handling of asynchronous event notifications

The events notification mechanism shall support the assigning of a user function as an event notification handler. Furthermore, the mechanism shall support the preemption of an event handler function by a higher priority event notification and shall support the selection of the highest priority pending event notification when multiple notifications (of different priority) are pending simultaneously.

The model here is based on hardware interrupts. Asynchronous event handling allows the application to ensure that time-critical events are immediately processed when delivered, without the indeterminism of being at a random location within a polling loop. Use of handler priority allows the specification of how handlers are interrupted by other higher priority handlers.

  - Differentiation between multiple occurrences of event notifications of the same type

    The events notification mechanism shall pass an application-defined value to the event handler function. This value can be used for a variety of purposes, such as enabling the application to identify which of several possible events of the same type (for example, timer expirations) has occurred.

  - Polled reception of asynchronous event notifications

    The events notification mechanism shall support blocking and non-blocking polls for asynchronous event notification.

    The polled mode of operation is often preferred over the interrupt mode by those practitioners accustomed to this model. Providing support for this model facilitates the porting of applications based on this model to POSIX.1b conforming systems.

  - Deterministic response to asynchronous event notifications

    The events notification mechanism shall not preclude implementations that provide deterministic event dispatch latency and shall minimize the number of system calls needed to use the event facilities during realtime processing.

- Rationale for Extension

  POSIX.1 signals have many of the characteristics necessary to support the asynchronous handling of event notifications, and the Realtime Signals Extension addresses the following deficiencies in the POSIX.1 signal mechanism:

  - Signals do not support reliable delivery of event notification. Subsequent occurrences of a pending signal are not guaranteed to be delivered.

  - Signals do not support prioritized delivery of event notifications. The order of signal delivery when multiple unblocked signals are pending is undefined.

  - Signals do not support the differentiation between multiple signals of the same type.

B.2.8.2 Asynchronous I/O

Many applications need to interact with the I/O subsystem in an asynchronous manner. The asynchronous I/O mechanism provides the ability to overlap application processing and I/O operations initiated by the application. The asynchronous I/O mechanism allows a single process to perform I/O simultaneously to a single file multiple times or to multiple files multiple times.

Overview

Asynchronous I/O operations proceed in logical parallel with the processing done by the application after the asynchronous I/O has been initiated. Other than this difference, asynchronous I/O behaves similarly to normal I/O using *read*( ), *write*( ), *lseek*( ), and *fsync*( ). The effect of issuing an asynchronous I/O request is as if a separate thread of execution were to perform atomically the implied *lseek*( ) operation, if any, and then the requested I/O operation (either *read*( ), *write*( ), or *fsync*( )). There is no seek implied with a call to *aio_fsync*( ). Concurrent asynchronous operations and synchronous operations applied to the same file update the file as if the I/O operations had proceeded serially.

When asynchronous I/O completes, a signal can be delivered to the application to indicate the completion of the I/O. This signal can be used to indicate that buffers and control blocks used for asynchronous I/O can be reused. Signal delivery is not required for an asynchronous operation and may be turned off on a per-operation basis by the application. Signals may also be synchronously polled using *aio_suspend*( ), *sigtimedwait*( ), or *sigwaitinfo*( ).

Normal I/O has a return value and an error status associated with it. Asynchronous I/O returns a value and an error status when the operation is first submitted, but that only relates to whether the operation was successfully queued up for servicing. The I/O operation itself also has a return status and an error value. To allow the application to retrieve the return status and the error value, functions are provided that, given the address of an asynchronous I/O control block, yield the return and error status associated with the operation. Until an asynchronous I/O operation is done, its error status shall be [EINPROGRESS]. Thus, an application can poll for completion of an asynchronous I/O operation by waiting for the error status to become equal to a value other than [EINPROGRESS]. The return status of an asynchronous I/O operation is undefined so long as the error status is equal to [EINPROGRESS].

Storage for asynchronous operation return and error status may be limited. Submission of asynchronous I/O operations may fail if this storage is exceeded. When an application retrieves the return status of a given asynchronous operation, therefore, any system-maintained storage used for this status and the error status may be reclaimed for use by other asynchronous operations.

Asynchronous I/O can be performed on file descriptors that have been enabled for POSIX.1b synchronized I/O. In this case, the I/O operation still occurs asynchronously, as defined herein; however, the asynchronous operation I/O in this case is not completed until the I/O has reached either the state of synchronized I/O data integrity completion or synchronized I/O file integrity completion, depending on the sort of synchronized I/O that is enabled on the file descriptor.

Models

Three models illustrate the use of asynchronous I/O: a journalization model, a data acquisition model, and a model of the use of asynchronous I/O in supercomputing applications.

- Journalization Model

  Many realtime applications perform low-priority journalizing functions. Journalizing requires that logging records be queued for output without blocking the initiating process.

- Data Acquisition Model

  A data acquisition process may also serve as a model. The process has two or more channels delivering intermittent data that must be read within a certain time. The process issues one asynchronous read on each channel. When one of the channels needs data collection, the process reads the data and posts it through an asynchronous write to secondary memory for future processing.

- Supercomputing Model

  The supercomputing community has used asynchronous I/O much like that specified herein for many years. This community requires the ability to perform multiple I/O operations to multiple devices with a minimal number of entries to ‘‘the system’’; each entry to ‘‘the system’’ provokes a major delay in operations when compared to the normal progress made by the application. This existing practice motivated the use of combined *lseek*( ) and *read*( ) or *write*( ) calls, as well as the *lio_listio*( ) call. Another common practice is to disable signal notification for I/O completion, and simply poll for I/O completion at some interval by which the I/O should be completed. Likewise, interfaces like *aio_cancel*( ) have been in successful commercial use for many years. Note also that an underlying implementation of asynchronous I/O will require the ability, at least internally, to cancel outstanding asynchronous I/O, at least when the process exits. (Consider an asynchronous read from a terminal, when the process intends to exit immediately.)

Requirements

Asynchronous input and output for realtime implementations have these requirements:

- The ability to queue multiple asynchronous read and write operations to a single open instance. Both sequential and random access should be supported.
- The ability to queue asynchronous read and write operations to multiple open instances.
- The ability to obtain completion status information by polling and/or asynchronous event notification.
- Asynchronous event notification on asynchronous I/O completion is optional.
- It has to be possible for the application to associate the event with the *aiocbp* for the operation that generated the event.
- The ability to cancel queued requests.
- The ability to wait upon asynchronous I/O completion in conjunction with other types of events.
- The ability to accept an *aio_read*( ) and an *aio_cancel*( ) for a device that accepts a *read*( ), and the ability to accept an *aio_write*( ) and an *aio_cancel*( ) for a device that accepts a *write*( ). This does not imply that the operation is asynchronous.

Standardization Issues

The following issues are addressed by the standardization of asynchronous I/O:

- Rationale for New Interface

  Non-blocking I/O does not satisfy the needs of either realtime or high-performance computing models; these models require that a process overlap program execution and I/O processing. Realtime applications will often make use of direct I/O to or from the address space of the process, or require synchronized (unbuffered) I/O; they also require the ability to overlap this I/O with other computation. In addition, asynchronous I/O allows an application to keep a device busy at all times, possibly achieving greater throughput. Supercomputing and database architectures will often have specialized hardware that can provide true asynchrony underlying the logical asynchrony provided by this interface. In addition, asynchronous I/O should be supported by all types of files and devices in the same manner.

- Effect of Buffering

  If asynchronous I/O is performed on a file that is buffered prior to being actually written to the device, it is possible that asynchronous I/O will offer no performance advantage over normal I/O; the cycles *stolen* to perform the asynchronous I/O will be taken away from the running process and the I/O will occur at interrupt time. This potential lack of gain in performance in no way obviates the need for asynchronous I/O by realtime applications, which very often will use specialized hardware support; multiple processors; and/or unbuffered, synchronized I/O.

B.2.8.3 Memory Management

All memory management and shared memory definitions are located in the **<sys/mman.h>** header. This is for alignment with historical practice.

Memory Locking Functions

This portion of the rationale presents models, requirements, and standardization issues relevant to process memory locking.

- Models

  Realtime systems that conform to IEEE Std 1003.1-200x are expected (and desired) to be supported on systems with demand-paged virtual memory management, non-paged swapping memory management, and physical memory systems with no memory management hardware. The general case, however, is the demand-paged, virtual memory system with each POSIX process running in a virtual address space. Note that this includes architectures where each process resides in its own virtual address space and architectures where the address space of each process is only a portion of a larger global virtual address space.

  The concept of memory locking is introduced to eliminate the indeterminacy introduced by paging and swapping, and to support an upper bound on the time required to access the memory mapped into the address space of a process. Ideally, this upper bound will be the same as the time required for the processor to access ‘‘main memory’’, including any address translation and cache miss overheads. But some implementations—primarily on mainframes—will not actually force locked pages to be loaded and held resident in main memory. Rather, they will handle locked pages so that accesses to these pages will meet the performance metrics for locked process memory in the implementation. Also, although it is not, for example, the intention that this interface, as specified, be used to lock process memory into ‘‘cache’’, it is conceivable that an implementation could support a large static RAM memory and define this as ‘‘main memory’’ and use a large[r] dynamic RAM as ‘‘backing store’’. These interfaces could then be interpreted as supporting the locking of process memory into the static RAM. Support for multiple levels of backing store would require extensions to these interfaces.

  Implementations may also use memory locking to guarantee a fixed translation between virtual and physical addresses where such is beneficial to improving determinancy for direct-to/from-process input/output. IEEE Std 1003.1-200x does not guarantee to the application that the virtual-to-physical address translations, if such exist, are fixed, because such behavior would not be implementable on all architectures on which implementations of IEEE Std 1003.1-200x are expected. But IEEE Std 1003.1-200x does mandate that an implementation define, for the benefit of potential users, whether or not locking guarantees fixed translations.

  Memory locking is defined with respect to the address space of a process. Only the pages mapped into the address space of a process may be locked by the process, and when the pages are no longer mapped into the address space—for whatever reason—the locks established with respect to that address space are removed. Shared memory areas warrant special mention, as they may be mapped into more than one address space or mapped more than once into the address space of a process; locks may be established on pages within these areas with respect to several of these mappings. In such a case, the lock state of the underlying physical pages is the logical OR of the lock state with respect to each of the mappings. Only when all such locks have been removed are the shared pages considered unlocked.

In recognition of the page granularity of Memory Management Units (MMU), and in order to support locking of ranges of address space, memory locking is defined in terms of ''page'' granularity. That is, for the interfaces that support an address and size specification for the region to be locked, the address must be on a page boundary, and all pages mapped by the specified range are locked, if valid. This means that the length is implicitly rounded up to a multiple of the page size. The page size is implementation-defined and is available to applications as a compile time symbolic constant or at runtime via *sysconf*( ).

A ''real memory'' POSIX.1b implementation that has no MMU could elect not to support these interfaces, returning [ENOSYS]. But an application could easily interpret this as meaning that the implementation would unconditionally page or swap the application when such is not the case. It is the intention of IEEE Std 1003.1-200x that such a system could define these interfaces as ''NO-OPs'', returning success without actually performing any function except for mandated argument checking.

- Requirements

For realtime applications, memory locking is generally considered to be required as part of application initialization. This locking is performed after an application has been loaded (that is, *exec*'d) and the program remains locked for its entire lifetime. But to support applications that undergo major mode changes where, in one mode, locking is required, but in another it is not, the specified interfaces allow repeated locking and unlocking of memory within the lifetime of a process.

When a realtime application locks its address space, it should not be necessary for the application to then ''touch'' all of the pages in the address space to guarantee that they are resident or else suffer potential paging delays the first time the page is referenced. Thus, IEEE Std 1003.1-200x requires that the pages locked by the specified interfaces be resident when the locking functions return successfully.

Many architectures support system-managed stacks that grow automatically when the current extent of the stack is exceeded. A realtime application has a requirement to be able to ''preallocate'' sufficient stack space and lock it down so that it will not suffer page faults to grow the stack during critical realtime operation. There was no consensus on a portable way to specify how much stack space is needed, so IEEE Std 1003.1-200x supports no specific interface for preallocating stack space. But an application can portably lock down a specific amount of stack space by specifying MCL_FUTURE in a call to *memlockall*( ) and then calling a dummy function that declares an automatic array of the desired size.

Memory locking for realtime applications is also generally considered to be an ''all or nothing'' proposition. That is, the entire process, or none, is locked down. But, for applications that have well-defined sections that need to be locked and others that do not, IEEE Std 1003.1-200x supports an optional set of interfaces to lock or unlock a range of process addresses. Reasons for locking down a specific range include:

— An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated

— An input/output ''buffer'' area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated

Finally, locking is generally viewed as an ''application-wide'' function. That is, the application is globally aware of which regions are locked and which are not over time. This is in contrast to a function that is used temporarily within a ''third party'' library routine whose function is unknown to the application, and therefore must have no ''side effects''. The specified interfaces, therefore, do not support ''lock stacking'' or ''lock nesting'' within a process. But, for pages that are shared between processes or mapped more than once into a process address space, ''lock stacking'' is essentially mandated by the requirement that unlocking of pages that are mapped by more that one process or more than once by the same process does not affect locks established on the other mappings.

There was some support for ''lock stacking'' so that locking could be transparently used in functions or opaque modules. But the consensus was not to burden all implementations with lock stacking (and reference counting), and an implementation option was proposed. There were strong objections to the option because applications would have to support both options in order to remain portable. The consensus was to eliminate lock stacking altogether, primarily through overwhelming support for the System V ''m[un]lock[all]'' interface on which IEEE Std 1003.1-200x is now based.

Locks are not inherited across *fork*( )s because some implementations implement *fork*( ) by creating new address spaces for the child. In such an implementation, requiring locks to be inherited would lead to new situations in which a fork would fail due to the inability of the system to lock sufficient memory to lock both the parent and the child. The consensus was that there was no benefit to such inheritance. Note that this does not mean that locks are removed when, for instance, a thread is created in the same address space. |

Similarly, locks are not inherited across *exec* because some implementations implement *exec* by unmapping all of the pages in the address space (which, by definition, removes the locks on these pages), and maps in pages of the *exec*'d image. In such an implementation, requiring locks to be inherited would lead to new situations in which *exec* would fail. Reporting this failure would be very cumbersome to detect in time to report to the calling process, and no appropriate mechanism exists for informing the *exec*'d process of its status. |

It was determined that, if the newly loaded application required locking, it was the responsibility of that application to establish the locks. This is also in keeping with the general view that it is the responsibility of the application to be aware of all locks that are established.

There was one request to allow (not mandate) locks to be inherited across *fork*( ), and a request for a flag, MCL_INHERIT, that would specify inheritance of memory locks across *exec*s. Given the difficulties raised by this and the general lack of support for the feature in IEEE Std 1003.1-200x, it was not added. IEEE Std 1003.1-200x does not preclude an implementation from providing this feature for administrative purposes, such as a ''run'' command that will lock down and execute specified program. Additionally, the rationale for the objection equated *fork*( ) with creating a thread in the address space. IEEE Std 1003.1-200x does not mandate releasing locks when creating additional threads in an existing process.

- Standardization Issues

One goal of IEEE Std 1003.1-200x is to define a set of primitives that provide the necessary functionality for realtime applications, with consideration for the needs of other application domains where such were identified, which is based to the extent possible on existing industry practice.

The Memory Locking option is required by many realtime applications to tune performance. Such a facility is accomplished by placing constraints on the virtual memory system to limit paging of time of the process or of critical sections of the process. This facility should not be used by most non-realtime applications.

Optional features provided in IEEE Std 1003.1-200x allow applications to lock selected address ranges with the caveat that the process is responsible for being aware of the page granularity of locking and the unnested nature of the locks.

Mapped Files Functions

The Memory Mapped Files option provides a mechanism that allows a process to access files by directly incorporating file data into its address space. Once a file is ‘‘mapped’’ into a process address space, the data can be manipulated by instructions as memory. The use of mapped files can significantly reduce I/O data movement since file data does not have to be copied into process data buffers as in *read*( ) and *write*( ). If more than one process maps a file, its contents are shared among them. This provides a low overhead mechanism by which processes can synchronize and communicate.

- Historical Perspective

  Realtime applications have historically been implemented using a collection of cooperating processes or tasks. In early systems, these processes ran on bare hardware (that is, without an operating system) with no memory relocation or protection. The application paradigms that arose from this environment involve the sharing of data between the processes.

  When realtime systems were implemented on top of vendor-supplied operating systems, the paradigm or performance benefits of direct access to data by multiple processes was still deemed necessary. As a result, operating systems that claim to support realtime applications must support the shared memory paradigm.

  Additionally, a number of realtime systems provide the ability to map specific sections of the physical address space into the address space of a process. This ability is required if an application is to obtain direct access to memory locations that have specific properties (for example, refresh buffers or display devices, dual ported memory locations, DMA target locations). The use of this ability is common enough to warrant some degree of standardization of its interface. This ability overlaps the general paradigm of shared memory in that, in both instances, common global objects are made addressable by individual processes or tasks.

  Finally, a number of systems also provide the ability to map process addresses to files. This provides both a general means of sharing persistent objects, and using files in a manner that optimizes memory and swapping space usage.

  Simple shared memory is clearly a special case of the more general file mapping capability. In addition, there is relatively widespread agreement and implementation of the file mapping interface. In these systems, many different types of objects can be mapped (for example, files, memory, devices, and so on) using the same mapping interfaces. This approach both minimizes interface proliferation and maximizes the generality of programs using the mapping interfaces.

- Memory Mapped Files Usage

  A memory object can be concurrently mapped into the address space of one or more processes. The *mmap*( ) and *munmap*( ) functions allow a process to manipulate their address space by mapping portions of memory objects into it and removing them from it. When multiple processes map the same memory object, they can share access to the underlying data. Implementations may restrict the size and alignment of mappings to be on *page*-size boundaries. The page size, in bytes, is the value of the system-configurable variable {PAGESIZE}, typically accessed by calling *sysconf*( ) with a *name* argument of _SC_PAGESIZE. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size.

  To map memory, a process first opens a memory object. The *ftruncate*( ) function can be used to contract or extend the size of the memory object even when the object is currently mapped. If the memory object is extended, the contents of the extended areas are zeros.

After opening a memory object, the application maps the object into its address space using the *mmap*( ) function call. Once a mapping has been established, it remains mapped until unmapped with *munmap*( ), even if the memory object is closed. The *mprotect*( ) function can be used to change the memory protections initially established by *mmap*( ).

A *close*( ) of the file descriptor, while invalidating the file descriptor itself, does not unmap any mappings established for the memory object. The address space, including all mapped regions, is inherited on *fork*( ). The entire address space is unmapped on process termination or by successful calls to any of the *exec* family of functions.

The *msync*( ) function is used to force mapped file data to permanent storage.

- Effects on Other Functions

When the Memory Mapped Files option is supported, the operation of the *open*( ), *creat*( ), and *unlink*( ) functions are a natural result of using the file system name space to map the global names for memory objects.

The *ftruncate*( ) function can be use to set the length of a sharable memory object.

The meaning of *stat*( ) fields other than the size and protection information is undefined on implementations where memory objects are not implemented using regular files. When regular files are used, the times reflect when the implementation updated the file image of the data, not when a process updated the data in memory.

The operations of *fdopen*( ), *write*( ), *read*( ), and *lseek*( ) were made unspecified for objects opened with *shm_open*( ), so that implementations that did not implement memory objects as regular files would not have to support the operation of these functions on shared memory objects.

The behavior of memory objects with respect to *close*( ), *dup*( ), *dup2*( ), *open*( ), *close*( ), *fork*( ), *_exit*( ), and the *exec* family of functions is the same as the behavior of the existing practice of the *mmap*( ) function.

A memory object can still be referenced after a close. That is, any mappings made to the file are still in effect, and reads and writes that are made to those mappings are still valid and are shared with other processes that have the same mapping. Likewise, the memory object can still be used if any references remain after its name(s) have been deleted. Any references that remain after a close must not appear to the application as file descriptors.

This is existing practice for *mmap*( ) and *close*( ). In addition, there are already mappings present (text, data, stack) that do not have open file descriptors. The text mapping in particular is considered a reference to the file containing the text. The desire was to treat all mappings by the process uniformly. Also, many modern implementations use *mmap*( ) to implement shared libraries, and it would not be desirable to keep file descriptors for each of the many libraries an application can use. It was felt there were many other existing programs that used this behavior to free a file descriptor, and thus IEEE Std 1003.1-200x could not forbid it and still claim to be using existing practice.

For implementations that implement memory objects using memory only, memory objects will retain the memory allocated to the file after the last close and will use that same memory on the next open. Note that closing the memory object is not the same as deleting the name, since the memory object is still defined in the memory object name space.

The locks of *fcntl*( ) do not block any read or write operation, including read or write access to shared memory or mapped files. In addition, implementations that only support shared memory objects should not be required to implement record locks. The reference to *fcntl*( ) is added to make this point explicitly. The other *fcntl*( ) commands are useful with shared memory objects.

The size of pages that mapping hardware may be able to support may be a configurable value, or it may change based on hardware implementations. The addition of the _SC_PAGESIZE parameter to the *sysconf*( ) function is provided for determining the mapping page size at runtime.

Shared Memory Functions

Implementations may support the Shared Memory Objects option without supporting a general Memory Mapped Files option. Shared memory objects are named regions of storage that may be independent of the file system and can be mapped into the address space of one or more processes to allow them to share the associated memory.

- Requirements

  Shared memory is used to share data among several processes, each potentially running at different priority levels, responding to different inputs, or performing separate tasks. Shared memory is not just simply providing common access to data, it is providing the fastest possible communication between the processes. With one memory write operation, a process can pass information to as many processes as have the memory region mapped.

  As a result, shared memory provides a mechanism that can be used for all other interprocess communications facilities. It may also be used by an application for implementing more sophisticated mechanisms than semaphores and message queues.

  The need for a shared memory interface is obvious for virtual memory systems, where the operating system is directly preventing processes from accessing each other's data. However, in unprotected systems, such as those found in some embedded controllers, a shared memory interface is needed to provide a portable mechanism to allocate a region of memory to be shared and then to communicate the address of that region to other processes.

  This, then, provides the minimum functionality that a shared memory interface must have in order to support realtime applications: to allocate and name an object to be mapped into memory for potential sharing (*open*( ) or *shm_open*( )), and to make the memory object available within the address space of a process (*mmap*( )). To complete the interface, a mechanism to release the claim of a process on a shared memory object (*munmap*( )) is also needed, as well as a mechanism for deleting the name of a sharable object that was previously created (*unlink*( ) or *shm_unlink*( )).

  After a mapping has been established, an implementation should not have to provide services to maintain that mapping. All memory writes into that area will appear immediately in the memory mapping of that region by any other processes.

  Thus, requirements include:

  — Support creation of sharable memory objects and the mapping of these objects into the address space of a process.

  — Sharable memory objects should be accessed by global names accessible from all processes.

  — Support the mapping of specific sections of physical address space (such as a memory mapped device) into the address space of a process. This should not be done by the process specifying the actual address, but again by an implementation-defined global name (such as a special device name) dedicated to this purpose.

  — Support the mapping of discrete portions of these memory objects.

- — Support for minimum hardware configurations that contain no physical media on which to store shared memory contents permanently.
- — The ability to preallocate the entire shared memory region so that minimum hardware configurations without virtual memory support can guarantee contiguous space.
- — The maximizing of performance by not requiring functionality that would require implementation interaction above creating the shared memory area and returning the mapping.

Note that the above requirements do not preclude:

- — The sharable memory object from being implemented using actual files on an actual file system.
- — The global name that is accessible from all processes being restricted to a file system area that is dedicated to handling shared memory.
- — An implementation not providing implementation-defined global names for the purpose of physical address mapping.

- Shared Memory Objects Usage

If the Shared Memory Objects option is supported, a shared memory object may be created, or opened if it already exists, with the *shm_open*( ) function. If the shared memory object is created, it has a length of zero. The *ftruncate*( ) function can be used to set the size of the shared memory object after creation. The *shm_unlink*( ) function removes the name for a shared memory object created by *shm_open*( ).

- Shared Memory Overview

The shared memory facility defined by IEEE Std 1003.1-200x usually results in memory locations being added to the address space of the process. The implementation returns the address of the new space to the application by means of a pointer. This works well in languages like C. However, in languages without pointer types it will not work. In the bindings for such a language, either a special COMMON section will need to be defined (which is unlikely), or the binding will have to allow existing structures to be mapped. The implementation will likely have to place restrictions on the size and alignment of such structures or will have to map a suitable region of the address space of the process into the memory object, and thus into other processes. These are issues for that particular language binding. For IEEE Std 1003.1-200x, however, the practice will not be forbidden, merely undefined.

Two potentially different name spaces are used for naming objects that may be mapped into process address spaces. When the Memory Mapped Files option is supported, files may be accessed via *open*( ). When the Shared Memory Objects option is supported, sharable memory objects that might not be files may be accessed via the *shm_open*( ) function. These options are not mutually-exclusive.

Some implementations supporting the Shared Memory Objects option may choose to implement the shared memory object name space as part of the file system name space. There are several reasons for this:

- — It allows applications to prevent name conflicts by use of the directory structure.
- — It uses an existing mechanism for accessing global objects and prevents the creation of a new mechanism for naming global objects.

In such implementations, memory objects can be implemented using regular files, if that is what the implementation chooses. The *shm_open*( ) function can be implemented as an *open*( )

call in a fixed directory followed by a call to *fcntl*( ) to set FD_CLOEXEC. The *shm_unlink*( ) function can be implemented as an *unlink*( ) call.

On the other hand, it is also expected that small embedded systems that support the Shared Memory Objects option may wish to implement shared memory without having any file systems present. In this case, the implementations may choose to use a simple string valued name space for shared memory regions. The *shm_open*( ) function permits either type of implementation.

Some implementations have hardware that supports protection of mapped data from certain classes of access and some do not. Systems that supply this functionality can support the Memory Protection option.

Some implementations restrict size, alignment, and protections to be on *page*-size boundaries. If an implementation has no restrictions on size or alignment, it may specify a 1-byte page size. Applications on implementations that do support larger pages must be cognizant of the page size since this is the alignment and protection boundary.

Simple embedded implementations may have a 1-byte page size and only support the Shared Memory Objects option. This provides simple shared memory between processes without requiring mapping hardware.

IEEE Std 1003.1-200x specifically allows a memory object to remain referenced after a close because that is existing practice for the *mmap*( ) function.

Typed Memory Functions

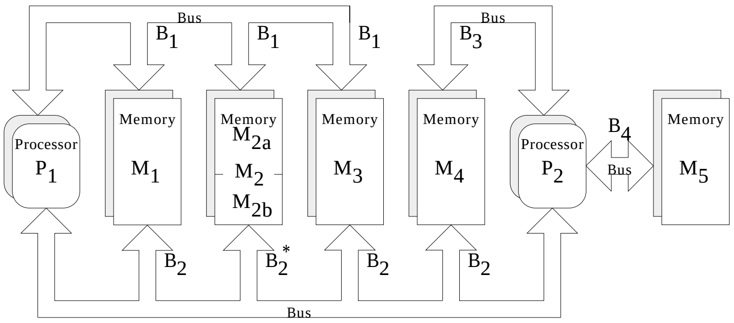
Implementations may support the Typed Memory Objects option without supporting either the Shared Memory option or the Memory Mapped Files option. Typed memory objects are pools of specialized storage, different from the main memory resource normally used by a processor to hold code and data, that can be mapped into the address space of one or more processes.

- Model

  Realtime systems conforming to one of the POSIX.13 realtime profiles are expected (and desired) to be supported on systems with more than one type or pool of memory (for example, SRAM, DRAM, ROM, EPROM, EEPROM), where each type or pool of memory may be accessible by one or more processors via one or more busses (ports). Memory mapped files, shared memory objects, and the language-specific storage allocation operators (*malloc*( ) for the ISO C standard, *new* for ISO Ada) fail to provide application program interfaces versatile enough to allow applications to control their utilization of such diverse memory resources. The typed memory interfaces *posix_typed_mem_open*( ), *posix_mem_offset*( ), *posix_typed_mem_get_info*( ), *mmap*( ), and *munmap*( ) defined herein support the model of typed memory described below.

  For purposes of this model, a system comprises several processors (for example, P1 and P2), several physical memory pools (for example, M1, M2, M2a, M2b, M3, M4, and M5), and several busses or ‘‘ports’’ (for example, B1, B2, B3, and B4) interconnecting the various processors and memory pools in some system-specific way. Notice that some memory pools may be contained in others (for example, M2a and M2b are contained in M2).

  Figure B-1 (on page 3412) shows an example of such a model. In a system like this, an application should be able to perform the following operations:



* All addresses in pool M_2 (comprising pools M_{2a} and M_{2b}) accessible via port B_1.
Addresses in pool M_{2b} are also accessible via port B_2.
Addresses in pool M_{2a} are *not* accessible via port B_2.

Figure B-1 Example of a System with Typed Memory

— Typed Memory Allocation

An application should be able to allocate memory dynamically from the desired pool using the desired bus, and map it into a process' address space. For example, processor P1 can allocate some portion of memory pool M1 through port B1, treating all unmapped subareas of M1 as a heap-storage resource from which memory may be allocated. This portion of memory is mapped into the process' address space, and subsequently deallocated when unmapped from all processes.

— Using the Same Storage Region from Different Busses

An application process with a mapped region of storage that is accessed from one bus should be able to map that same storage area at another address (subject to page size restrictions detailed in *mmap*( )), to allow it to be accessed from another bus. For example, processor P1 may wish to access the same region of memory pool M2b both through ports B1 and B2.

— Sharing Typed Memory Regions

Several application processes running on the same or different processors may wish to share a particular region of a typed memory pool. Each process or processor may wish to access this region through different busses. For example, processor P1 may want to share a region of memory pool M4 with processor P2, and they may be required to use busses B2 and B3, respectively, to minimize bus contention. A problem arises here when a process allocates and maps a portion of fragmented memory and then wants to share this region of memory with another process, either in the same processor or different processors. The solution adopted is to allow the first process to find out the memory map (offsets and lengths) of all the different fragments of memory that were mapped into its address space, by repeatedly calling *posix_mem_offset*( ). Then, this process can pass the offsets and lengths obtained to the second process, which can then map the same memory fragments into its address space.

— Contiguous Allocation

The problem of finding the memory map of the different fragments of the memory pool that were mapped into logically contiguous addresses of a given process, can be solved by requesting contiguous allocation. For example, a process in P1 can allocate 10 Kbytes of physically contiguous memory from M3-B1, and obtain the offset (within pool M3) of this block of memory. Then, it can pass this offset (and the length) to a process in P2 using some interprocess communication mechanism. The second process can map the same block of memory by using the offset transferred and specifying M3-B2.

— Unallocated Mapping

Any subarea of a memory pool that is mapped to a process, either as the result of an allocation request or an explicit mapping, is normally unavailable for allocation. Special processes such as debuggers, however, may need to map large areas of a typed memory pool, yet leave those areas available for allocation.

Typed memory allocation and mapping has to coexist with storage allocation operators like *malloc*( ), but systems are free to choose how to implement this coexistence. For example, it may be system configuration-dependent if all available system memory is made part of one of the typed memory pools or if some part will be restricted to conventional allocation operators. Equally system configuration-dependent may be the availability of operators like *malloc*( ) to allocate storage from certain typed memory pools. It is not excluded to configure a system such that a given named pool, P1, is in turn split into non-overlapping named subpools. For example, M1-B1, M2-B1, and M3-B1 could also be accessed as one common pool M123-B1. A call to *malloc*( ) on P1 could work on such a larger pool while full optimization of memory usage by P1 would require typed memory allocation at the subpool level.

- Existing Practice

OS-9 provides for the naming (numbering) and prioritization of memory types by a system administrator. It then provides APIs to request memory allocation of typed (colored) memory by number, and to generate a bus address from a mapped memory address (translate). When requesting colored memory, the user can specify type 0 to signify allocation from the first available type in priority order.

HP-RT presents interfaces to map different kinds of storage regions that are visible through a VME bus, although it does not provide allocation operations. It also provides functions to perform address translation between VME addresses and virtual addresses. It represents a VME-bus unique solution to the general problem.

The PSOS approach is similar (that is, based on a pre-established mapping of bus address ranges to specific memories) with a concept of segments and regions (regions dynamically allocated from a heap which is a special segment). Therefore, PSOS does not fully address the general allocation problem either. PSOS does not have a ‘‘process’’-based model, but more of a ‘‘thread’’-only-based model of multi-tasking. So mapping to a process address space is not an issue.

QNX (a Canadian OS vendor specializing in realtime embedded systems on 80x86-based processors) uses the System V approach of opening specially named devices (shared memory segments) and using *mmap*( ) to then gain access from the process. They do not address allocation directly, but once typed shared memory can be mapped, an ‘‘allocation manager’’ process could be written to handle requests for allocation.

The System V approach also included allocation, implemented by opening yet other special ‘‘devices’’ which allocate, rather than appearing as a whole memory object.

The Orkid realtime kernel interface definition has operations to manage memory ‘‘regions’’ and ‘‘pools’’, which are areas of memory that may reflect the differing physical nature of the memory. Operations to allocate memory from these regions and pools are also provided.

- Requirements

  Existing practice in SVID-derived UNIX systems relies on functionality similar to *mmap*( ) and its related interfaces to achieve mapping and allocation of typed memory. However, the issue of sharing typed memory (allocated or mapped) and the complication of multiple ports are not addressed in any consistent way by existing UNIX system practice. Part of this functionality is existing practice in specialized realtime operating systems. In order to solidify the capabilities implied by the model above, the following requirements are imposed on the interface:

  — Identification of Typed Memory Pools and Ports

  All processes (running in all processors) in the system shall be able to identify a particular (system configured) typed memory pool accessed through a particular (system configured) port by a name. That name shall be a member of a name space common to all these processes, but need not be the same name space as that containing ordinary filenames. The association between memory pools/ports and corresponding names is typically established when the system is configured. The ''open'' operation for typed memory objects should be distinct from the *open*( ) function, for consistency with other similar services, but implementable on top of *open*( ). This implies that the handle for a typed memory object will be a file descriptor.

  — Allocation and Mapping of Typed Memory

  Once a typed memory object has been identified by a process, it shall be possible to both map user-selected subareas of that object into process address space and to map system-selected (that is, dynamically allocated) subareas of that object, with user-specified length, into process address space. It shall also be possible to determine the maximum length of memory allocation that may be requested from a given typed memory object.

  — Sharing Typed Memory

  Two or more processes shall be able to share portions of typed memory, either user-selected or dynamically allocated. This requirement applies also to dynamically allocated regions of memory that are composed of several non-contiguous pieces.

  — Contiguous Allocation

  For dynamic allocation, it shall be the user's option whether the system is required to allocate a contiguous subarea within the typed memory object, or whether it is permitted to allocate discontiguous fragments which appear contiguous in the process mapping. Contiguous allocation simplifies the process of sharing allocated typed memory, while discontiguous allocation allows for potentially better recovery of deallocated typed memory.

  — Accessing Typed Memory Through Different Ports

  Once a subarea of a typed memory object has been mapped, it shall be possible to determine the location and length corresponding to a user-selected portion of that object within the memory pool. This location and length can then be used to remap that portion of memory for access from another port. If the referenced portion of typed memory was allocated discontiguously, the length thus determined may be shorter than anticipated, and the user code shall adapt to the value returned.

  — Deallocation

  When a previously mapped subarea of typed memory is no longer mapped by any process in the system—as a result of a call or calls to *munmap*( )—that subarea shall become potentially reusable for dynamic allocation; actual reuse of the subarea is a function of the dynamic typed memory allocation policy.

— Unallocated Mapping

It shall be possible to map user-selected subareas of a typed memory object without marking that subarea as unavailable for allocation. This option is not the default behavior, and shall require appropriate privilege.

- Scenario

The following scenario will serve to clarify the use of the typed memory interfaces.

Process A running on P1 (see Figure B-1 (on page 3412)) wants to allocate some memory from memory pool M2, and it wants to share this portion of memory with process B running on P2. Since P2 only has access to the lower part of M2, both processes will use the memory pool named M2b which is the part of M2 that is accessible both from P1 and P2. The operations that both processes need to perform are shown below:

— Allocating Typed Memory

Process A calls *posix_typed_mem_open*( ) with the name **/typed.m2b-b1** and a *tflag* of POSIX_TYPED_MEM_ALLOCATE to get a file descriptor usable for allocating from pool M2b accessed through port B1. It then calls *mmap*( ) with this file descriptor requesting a length of 4 096 bytes. The system allocates two discontiguous blocks of sizes 1 024 and 3 072 bytes within M2b. The *mmap*( ) function returns a pointer to a 4 096 byte array in process A's logical address space, mapping the allocated blocks contiguously. Process A can then utilize the array, and store data in it.

— Determining the Location of the Allocated Blocks

Process A can determine the lengths and offsets (relative to M2b) of the two blocks allocated, by using the following procedure: First, process A calls *posix_mem_offset*( ) with the address of the first element of the array and length 4 096. Upon return, the offset and length (1 024 bytes) of the first block are returned. A second call to *posix_mem_offset*( ) is then made using the address of the first element of the array plus 1 024 (the length of the first block), and a new length of 4 096–1 024. If there were more fragments allocated, this procedure could have been continued within a loop until the offsets and lengths of all the blocks were obtained. Notice that this relatively complex procedure can be avoided if contiguous allocation is requested (by opening the typed memory object with the *tflag* POSIX_TYPED_MEM_ALLOCATE_CONTIG).

— Sharing Data Across Processes

Process A passes the two offset values and lengths obtained from the *posix_mem_offset*( ) calls to process B running on P2, via some form of interprocess communication. Process B can gain access to process A's data by calling *posix_typed_mem_open*( ) with the name **/typed.m2b-b2** and a *tflag* of zero, then using two *mmap*( ) calls on the resulting file descriptor to map the two subareas of that typed memory object to its own address space.

- Rationale for no *mem_alloc*( ) and *mem_free*( )

The standard developers had originally proposed a pair of new flags to *mmap*( ) which, when applied to a typed memory object descriptor, would cause *mmap*( ) to allocate dynamically from an unallocated and unmapped area of the typed memory object. Deallocation was similarly accomplished through the use of *munmap*( ). This was rejected by the ballot group because it excessively complicated the (already rather complex) *mmap*( ) interface and introduced semantics useful only for typed memory, to a function which must also map shared memory and files. They felt that a memory allocator should be built on top of *mmap*( ) instead of being incorporated within the same interface, much as the ISO C standard libraries build *malloc*( ) on top of the virtual memory mapping functions *brk*( ) and *sbrk*( ). This would eliminate the complicated semantics involved with unmapping only part of an allocated block of typed memory.

To attempt to achieve ballot group consensus, typed memory allocation and deallocation was first migrated from *mmap*( ) and *munmap*( ) to a pair of complementary functions modeled on the ISO C standard *malloc*( ) and *free*( ). The *mem_alloc*( ) function specified explicitly the typed memory object (typed memory pool/access port) from which allocation takes place, unlike *malloc*( ) where the memory pool and port are unspecified. The *mem_free*( ) function handled deallocation. These new semantics still met all of the requirements detailed above without modifying the behavior of *mmap*( ) except to allow it to map specified areas of typed memory objects. An implementation would have been free to implement *mem_alloc*( ) and *mem_free*( ) over *mmap*( ), through *mmap*( ), or independently but cooperating with *mmap*( ).

The ballot group was queried to see if this was an acceptable alternative, and while there was some agreement that it achieved the goal of removing the complicated semantics of allocation from the *mmap*( ) interface, several balloters realized that it just created two additional functions that behaved, in great part, like *mmap*( ). These balloters proposed an alternative which has been implemented here in place of a separate *mem_alloc*( ) and *mem_free*( ). This alternative is based on four specific suggestions:

1. The *posix_typed_mem_open*( ) function should provide a flag which specifies ‘‘allocate on *mmap*( )’’ (otherwise, *mmap*( ) just maps the underlying object). This allows things roughly similar to **/dev/zero** *versus* **/dev/swap**. Two such flags have been implemented, one of which forces contiguous allocation.

2. The *posix_mem_offset*( ) function is acceptable because it can be applied usefully to mapped objects in general. It should return the file descriptor of the underlying object.

3. The *mem_get_info*( ) function in an earlier draft should be renamed *posix_typed_mem_get_info*( ) because it is not generally applicable to memory objects. It should probably return the file descriptor's allocation attribute. We have implemented the renaming of the function, but reject having it return a piece of information which is readily known by an application without this function. Its whole purpose is to query the typed memory object for attributes that are not user-specified, but determined by the implementation.

4. There should be no separate *mem_alloc*( ) or *mem_free*( ) functions. Instead, using *mmap*( ) on a typed memory object opened with an ‘‘allocate on *mmap*( )’’ flag should be used to force allocation. These are precisely the semantics defined in the current draft.

- Rationale for no Typed Memory Access Management

The working group had originally defined an additional interface (and an additional kind of object: typed memory master) to establish and dissolve mappings to typed memory on behalf of devices or processors which were independent of the operating system and had no inherent capability to directly establish mappings on their own. This was to have provided functionality similar to device driver interfaces such as *physio*( ) and their underlying bus-specific interfaces (for example, *mballoc*( )) which serve to set up and break down DMA pathways, and derive mapped addresses for use by hardware devices and processor cards.

The ballot group felt that this was beyond the scope of POSIX.1 and its amendments. Furthermore, the removal of interrupt handling interfaces from a preceding amendment (the IEEE Std 1003.1d-1999) during its balloting process renders these typed memory access management interfaces an incomplete solution to portable device management from a user process; it would be possible to initiate a device transfer to/from typed memory, but impossible to handle the transfer-complete interrupt in a portable way.

To achieve ballot group consensus, all references to typed memory access management capabilities were removed. The concept of portable interfaces from a device driver to both operating system and hardware is being addressed by the Uniform Driver Interface (UDI) industry forum, with formal standardization deferred until proof of concept and industry-wide acceptance and implementation.

B.2.8.4 Process Scheduling

IEEE PASC Interpretation 1003.1 #96 has been applied, adding the *pthread_setschedprio*( ) function. This was added since previously there was no way for a thread to lower its own priority without going to the tail of the threads list for its new priority. This capability is necessary to bound the duration of priority inversion encountered by a thread.

The following portion of the rationale presents models, requirements, and standardization issues relevant to process scheduling; see also Section B.2.9.4 (on page 3456).

In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute to meet implementation-defined goals. For time-sharing systems, the goal is to enhance system throughput and promote fairness; the application is provided little or no control over this sequencing function. While this is acceptable and desirable behavior in a time-sharing system, it is inappropriate in a realtime system; realtime applications must specifically control the execution sequence of their concurrent processes in order to meet externally defined response requirements.

In IEEE Std 1003.1-200x, the control over process sequencing is provided using a concept of scheduling policies. These policies, described in detail in this section, define the behavior of the system whenever processor resources are to be allocated to competing processes. Only the behavior of the policy is defined; conforming implementations are free to use any mechanism desired to achieve the described behavior.

- Models

  In an operating system supporting multiple concurrent processes, the system determines the order in which processes execute and might force long-running processes to yield to other processes at certain intervals. Typically, the scheduling code is executed whenever an event occurs that might alter the process to be executed next.

  The simplest scheduling strategy is a ‘‘first-in, first-out’’ (FIFO) dispatcher. Whenever a process becomes runnable, it is placed on the end of a ready list. The process at the front of the ready list is executed until it exits or becomes blocked, at which point it is removed from the list. This scheduling technique is also known as ‘‘run-to-completion’’ or ‘‘run-to-block’’.

  A natural extension to this scheduling technique is the assignment of a ‘‘non-migrating priority’’ to each process. This policy differs from strict FIFO scheduling in only one respect: whenever a process becomes runnable, it is placed at the end of the list of processes runnable at that priority level. When selecting a process to run, the system always selects the first process from the highest priority queue with a runnable process. Thus, when a process becomes unblocked, it will preempt a running process of lower priority without otherwise altering the ready list. Further, if a process elects to alter its priority, it is removed from the ready list and reinserted, using its new priority, according to the policy above.

  While the above policy might be considered unfriendly in a time-sharing environment in which multiple users require more balanced resource allocation, it could be ideal in a realtime environment for several reasons. The most important of these is that it is deterministic: the highest-priority process is always run and, among processes of equal priority, the process that has been runnable for the longest time is executed first. Because of this determinism, cooperating processes can implement more complex scheduling simply by altering their priority. For instance, if processes at a single priority were to reschedule themselves at fixed time intervals, a time-slice policy would result.

In a dedicated operating system in which all processes are well-behaved realtime applications, non-migrating priority scheduling is sufficient. However, many existing implementations provide for more complex scheduling policies.

IEEE Std 1003.1-200x specifies a linear scheduling model. In this model, every process in the system has a priority. The system scheduler always dispatches a process that has the highest (generally the most time-critical) priority among all runnable processes in the system. As long as there is only one such process, the dispatching policy is trivial. When multiple processes of equal priority are eligible to run, they are ordered according to a strict run-to-completion (FIFO) policy.

The priority is represented as a positive integer and is inherited from the parent process. For processes running under a fixed priority scheduling policy, the priority is never altered except by an explicit function call.

It was determined arbitrarily that larger integers correspond to ‘‘higher priorities’’.

Certain implementations might impose restrictions on the priority ranges to which processes can be assigned. There also can be restrictions on the set of policies to which processes can be set.

- Requirements

Realtime processes require that scheduling be fast and deterministic, and that it guarantees to preempt lower priority processes.

Thus, given the linear scheduling model, realtime processes require that they be run at a priority that is higher than other processes. Within this framework, realtime processes are free to yield execution resources to each other in a completely portable and implementation-defined manner.

As there is a generally perceived requirement for processes at the same priority level to share processor resources more equitably, provisions are made by providing a scheduling policy (that is, SCHED_RR) intended to provide a timeslice-like facility.

Note: The following topics assume that low numeric priority implies low scheduling criticality and *vice versa.*

- Rationale for New Interface

Realtime applications need to be able to determine when processes will run in relation to each other. It must be possible to guarantee that a critical process will run whenever it is runnable; that is, whenever it wants to for as long as it needs. SCHED_FIFO satisfies this requirement. Additionally, SCHED_RR was defined to meet a realtime requirement for a well-defined time-sharing policy for processes at the same priority.

It would be possible to use the BSD *setpriority*( ) and *getpriority*( ) functions by redefining the meaning of the ‘‘nice’’ parameter according to the scheduling policy currently in use by the process. The System V *nice*( ) interface was felt to be undesirable for realtime because it specifies an adjustment to the ‘‘nice’’ value, rather than setting it to an explicit value. Realtime applications will usually want to set priority to an explicit value. Also, System V *nice*( ) does not allow for changing the priority of another process.

With the POSIX.1b interfaces, the traditional ‘‘nice’’ value does not affect the SCHED_FIFO or SCHED_RR scheduling policies. If a ‘‘nice’’ value is supported, it is implementation-defined whether it affects the SCHED_OTHER policy.

An important aspect of IEEE Std 1003.1-200x is the explicit description of the queuing and preemption rules. It is critical, to achieve deterministic scheduling, that such rules be stated clearly in IEEE Std 1003.1-200x.

IEEE Std 1003.1-200x does not address the interaction between priority and swapping. The issues involved with swapping and virtual memory paging are extremely implementation-defined and would be nearly impossible to standardize at this point. The proposed scheduling paradigm, however, fully describes the scheduling behavior of runnable processes, of which one criterion is that the working set be resident in memory. Assuming the existence of a portable interface for locking portions of a process in memory, paging behavior need not affect the scheduling of realtime processes.

IEEE Std 1003.1-200x also does not address the priorities of ‘‘system’’ processes. In general, these processes should always execute in low-priority ranges to avoid conflict with other realtime processes. Implementations should document the priority ranges in which system processes run.

The default scheduling policy is not defined. The effect of I/O interrupts and other system processing activities is not defined. The temporary lending of priority from one process to another (such as for the purposes of affecting freeing resources) by the system is not addressed. Preemption of resources is not addressed. Restrictions on the ability of a process to affect other processes beyond a certain level (influence levels) is not addressed.

The rationale used to justify the simple time-quantum scheduler is that it is common practice to depend upon this type of scheduling to assure ‘‘fair’’ distribution of processor resources among portions of the application that must interoperate in a serial fashion. Note that IEEE Std 1003.1-200x is silent with respect to the setting of this time quantum, or whether it is a system-wide value or a per-process value, although it appears that the prevailing realtime practice is for it to be a system-wide value.

In a system with N processes at a given priority, all processor-bound, in which the time quantum is equal for all processes at a specific priority level, the following assumptions are made of such a scheduling policy:

1. A time quantum Q exists and the current process will own control of the processor for at least a duration of Q and will have the processor for a duration of Q.
2. The Nth process at that priority will control a processor within a duration of $(N-1) \times Q$.

These assumptions are necessary to provide equal access to the processor and bounded response from the application.

The assumptions hold for the described scheduling policy only if no system overhead, such as interrupt servicing, is present. If the interrupt servicing load is non-zero, then one of the two assumptions becomes fallacious, based upon how Q is measured by the system.

If Q is measured by clock time, then the assumption that the process obtains a duration Q processor time is false if interrupt overhead exists. Indeed, a scenario can be constructed with N processes in which a single process undergoes complete processor starvation if a peripheral device, such as an analog-to-digital converter, generates significant interrupt activity periodically with a period of $N \times Q$.

If Q is measured as actual processor time, then the assumption that the Nth process runs in within the duration $(N-1) \times Q$ is false.

It should be noted that SCHED_FIFO suffers from interrupt-based delay as well. However, for SCHED_FIFO, the implied response of the system is ‘‘as soon as possible’’, so that the interrupt load for this case is a vendor selection and not a compliance issue.

With this in mind, it is necessary either to complete the definition by including bounds on the interrupt load, or to modify the assumptions that can be made about the scheduling policy.

Since the motivation of inclusion of the policy is common usage, and since current applications do not enjoy the luxury of bounded interrupt load, item (2) above is sufficient to express existing application needs and is less restrictive in the standard definition. No difference in interface is necessary.

In an implementation in which the time quantum is equal for all processes at a specific priority, our assumptions can then be restated as:

- A time quantum Q exists, and a processor-bound process will be rescheduled after a duration of, at most, Q. Time quantum Q may be defined in either wall clock time or execution time.
- In general, the Nth process of a priority level should wait no longer than $(N-1) \times Q$ time to execute, assuming no processes exist at higher priority levels.
- No process should wait indefinitely.

For implementations supporting per-process time quanta, these assumptions can be readily extended.

Sporadic Server Scheduling Policy

The sporadic server is a mechanism defined for scheduling aperiodic activities in time-critical realtime systems. This mechanism reserves a certain bounded amount of execution capacity for processing aperiodic events at a high priority level. Any aperiodic events that cannot be processed within the bounded amount of execution capacity are executed in the background at a low priority level. Thus, a certain amount of execution capacity can be guaranteed to be available for processing periodic tasks, even under burst conditions in the arrival of aperiodic processing requests (that is, a large number of requests in a short time interval). The sporadic server also simplifies the schedulability analysis of the realtime system, because it allows aperiodic processes or threads to be treated as if they were periodic. The sporadic server was first described by Sprunt, et al.

The key concept of the sporadic server is to provide and limit a certain amount of computation capacity for processing aperiodic events at their assigned normal priority, during a time interval called the *replenishment period*. Once the entity controlled by the sporadic server mechanism is initialized with its period and execution-time budget attributes, it preserves its execution capacity until an aperiodic request arrives. The request will be serviced (if there are no higher priority activities pending) as long as there is execution capacity left. If the request is completed, the actual execution time used to service it is subtracted from the capacity, and a replenishment of this amount of execution time is scheduled to happen one replenishment period after the arrival of the aperiodic request. If the request is not completed, because there is no execution capacity left, then the aperiodic process or thread is assigned a lower background priority. For each portion of consumed execution capacity the execution time used is replenished after one replenishment period. At the time of replenishment, if the sporadic server was executing at a background priority level, its priority is elevated to the normal level. Other similar replenishment policies have been defined, but the one presented here represents a compromise between efficiency and implementation complexity.

The interface that appears in this section defines a new scheduling policy for threads and processes that behaves according to the rules of the sporadic server mechanism. Scheduling attributes are defined and functions are provided to allow the user to set and get the parameters that control the scheduling behavior of this mechanism, namely the normal and low priority, the replenishment period, the maximum number of pending replenishment operations, and the initial execution-time budget.

- Scheduling Aperiodic Activities

  Virtually all realtime applications are required to process aperiodic activities. In many cases, there are tight timing constraints that the response to the aperiodic events must meet. Usual timing requirements imposed on the response to these events are:

  — The effects of an aperiodic activity on the response time of lower priority activities must be controllable and predictable.

  — The system must provide the fastest possible response time to aperiodic events.

  — It must be possible to take advantage of all the available processing bandwidth not needed by time-critical activities to enhance average-case response times to aperiodic events.

  Traditional methods for scheduling aperiodic activities are background processing, polling tasks, and direct event execution:

  — Background processing consists of assigning a very low priority to the processing of aperiodic events. It utilizes all the available bandwidth in the system that has not been consumed by higher priority threads. However, it is very difficult, or impossible, to meet requirements on average-case response time, because the aperiodic entity has to wait for the execution of all other entities which have higher priority.

  — Polling consists of creating a periodic process or thread for servicing aperiodic requests. At regular intervals, the polling entity is started and it services accumulated pending aperiodic requests. If no aperiodic requests are pending, the polling entity suspends itself until its next period. Polling allows the aperiodic requests to be processed at a higher priority level. However, worst and average-case response times of polling entities are a direct function of the polling period, and there is execution overhead for each polling period, even if no event has arrived. If the deadline of the aperiodic activity is short compared to the inter-arrival time, the polling frequency must be increased to guarantee meeting the deadline. For this case, the increase in frequency can dramatically reduce the efficiency of the system and, therefore, its capacity to meet all deadlines. Yet, polling represents a good way to handle a large class of practical problems because it preserves system predictability, and because the amortized overhead drops as load increases.

  — Direct event execution consists of executing the aperiodic events at a high fixed-priority level. Typically, the aperiodic event is processed by an interrupt service routine as soon as it arrives. This technique provides predictable response times for aperiodic events, but makes the response times of all lower priority activities completely unpredictable under burst arrival conditions. Therefore, if the density of aperiodic event arrivals is unbounded, it may be a dangerous technique for time-critical systems. Yet, for those cases in which the physics of the system imposes a bound on the event arrival rate, it is probably the most efficient technique.

  — The sporadic server scheduling algorithm combines the predictability of the polling approach with the short response times of the direct event execution. Thus, it allows systems to meet an important class of application requirements that cannot be met by using the traditional approaches. Multiple sporadic servers with different attributes can be applied to the scheduling of multiple classes of aperiodic events, each with different kinds of timing requirements, such as individual deadlines, average response times, and so on. It also has many other interesting applications for realtime, such as scheduling producer/consumer tasks in time-critical systems, limiting the effects of faults on the estimation of task execution-time requirements, and so on.

- Existing Practice

  The sporadic server has been used in different kinds of applications, including military avionics, robot control systems, industrial automation systems, and so on. There are examples of many systems that cannot be successfully scheduled using the classic approaches, such as direct event execution, or polling, and are schedulable using a sporadic server scheduler. The sporadic server algorithm itself can successfully schedule all systems scheduled with direct event execution or polling.

  The sporadic server scheduling policy has been implemented as a commercial product in the run-time system of the Verdix Ada compiler. There are also many applications that have used a much less efficient application-level sporadic server. These real-time applications would benefit from a sporadic server scheduler implemented at the scheduler level.

- Library-Level *versus* Kernel-Level Implementation

  The sporadic server interface described in this section requires the sporadic server policy to be implemented at the same level as the scheduler. This means that the process sporadic server shall be implemented at the kernel level and the thread sporadic server policy shall be implemented at the same level as the thread scheduler; that is, kernel or library level.

  In an earlier interface for the sporadic server, this mechanism was implementable at a different level than the scheduler. This feature allowed the implementer to choose between an efficient scheduler-level implementation, or a simpler user or library-level implementation. However, the working group considered that this interface made the use of sporadic servers more complex, and that library-level implementations would lack some of the important functionality of the sporadic server, namely the limitation of the actual execution time of aperiodic activities. The working group also felt that the interface described in this chapter does not preclude library-level implementations of threads intended to provide efficient low-overhead scheduling for those threads that are not scheduled under the sporadic server policy.

- Range of Scheduling Priorities

  Each of the scheduling policies supported in IEEE Std 1003.1-200x has an associated range of priorities. The priority ranges for each policy might or might not overlap with the priority ranges of other policies. For time-critical realtime applications it is usual for periodic and aperiodic activities to be scheduled together in the same processor. Periodic activities will usually be scheduled using the SCHED_FIFO scheduling policy, while aperiodic activities may be scheduled using SCHED_SPORADIC. Since the application developer will require complete control over the relative priorities of these activities in order to meet his timing requirements, it would be desirable for the priority ranges of SCHED_FIFO and SCHED_SPORADIC to overlap completely. Therefore, although IEEE Std 1003.1-200x does not require any particular relationship between the different priority ranges, it is recommended that these two ranges should coincide.

- Dynamically Setting the Sporadic Server Policy

  Several members of the working group requested that implementations should not be required to support dynamically setting the sporadic server scheduling policy for a thread. The reason is that this policy may have a high overhead for library-level implementations of threads, and if threads are allowed to dynamically set this policy, this overhead can be experienced even if the thread does not use that policy. By disallowing the dynamic setting of the sporadic server scheduling policy, these implementations can accomplish efficient scheduling for threads using other policies. If a strictly conforming application needs to use the sporadic server policy, and is therefore willing to pay the overhead, it must set this policy at the time of thread creation.

- Limitation of the Number of Pending Replenishments

  The number of simultaneously pending replenishment operations must be limited for each sporadic server for two reasons: an unlimited number of replenishment operations would need an unlimited number of system resources to store all the pending replenishment operations; on the other hand, in some implementations each replenishment operation will represent a source of priority inversion (just for the duration of the replenishment operation) and thus, the maximum amount of replenishments must be bounded to guarantee bounded response times. The way in which the number of replenishments is bounded is by lowering the priority of the sporadic server to *sched_ss_low_priority* when the number of pending replenishments has reached its limit. In this way, no new replenishments are scheduled until the number of pending replenishments decreases.

  In the sporadic server scheduling policy defined in IEEE Std 1003.1-200x, the application can specify the maximum number of pending replenishment operations for a single sporadic server, by setting the value of the *sched_ss_max_repl* scheduling parameter. This value must be between one and {SS_REPL_MAX}, which is a maximum limit imposed by the implementation. The limit {SS_REPL_MAX} must be greater than or equal to {_POSIX_SS_REPL_MAX}, which is defined to be four in IEEE Std 1003.1-200x. The minimum limit of four was chosen so that an application can at least guarantee that four different aperiodic events can be processed during each interval of length equal to the replenishment period.

B.2.8.5 Clocks and Timers

- Clocks

  IEEE Std 1003.1-200x and the ISO C standard both define functions for obtaining system time. Implicit behind these functions is a mechanism for measuring passage of time. This specification makes this mechanism explicit and calls it a clock. The CLOCK_REALTIME clock required by IEEE Std 1003.1-200x is a higher resolution version of the clock that maintains POSIX.1 system time. This is a ‘‘system-wide’’ clock, in that it is visible to all processes and, were it possible for multiple processes to all read the clock at the same time, they would see the same value.

  An extensible interface was defined, with the ability for implementations to define additional clocks. This was done because of the observation that many realtime platforms support multiple clocks, and it was desired to fit this model within the standard interface. But implementation-defined clocks need not represent actual hardware devices, nor are they necessarily system-wide.

- Timers

  Two timer types are required for a system to support realtime applications:

  1. One-shot

     A one-shot timer is a timer that is armed with an initial expiration time, either relative to the current time or at an absolute time (based on some timing base, such as time in seconds and nanoseconds since the Epoch). The timer expires once and then is disarmed. With the specified facilities, this is accomplished by setting the *it_value* member of the *value* argument to the desired expiration time and the *it_interval* member to zero.

  2. Periodic

     A periodic timer is a timer that is armed with an initial expiration time, again either relative or absolute, and a repetition interval. When the initial expiration occurs, the timer is reloaded with the repetition interval and continues counting. With the specified facilities, this is accomplished by setting the *it_value* member of the *value* argument to the desired initial expiration time and the *it_interval* member to the desired repetition interval.

For both of these types of timers, the time of the initial timer expiration can be specified in two ways:

1. Relative (to the current time)
2. Absolute

- Examples of Using Realtime Timers

  In the diagrams below, *S* indicates a program schedule, *R* shows a schedule method request, and *E* suggests an internal operating system event.

  — Periodic Timer: Data Logging

  During an experiment, it might be necessary to log realtime data periodically to an internal buffer or to a mass storage device. With a periodic scheduling method, a logging module can be started automatically at fixed time intervals to log the data.

  Program schedule is requested every 10 seconds.

```
    R         S         S         S         S         S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

  [Time (in Seconds)]

  To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with the TIMER_ABSTIME flag reset, and with an initial expiration value and a repetition interval of 10 seconds.

  — One-shot Timer (Relative Time): Device Initialization

  In an emission test environment, large sample bags are used to capture the exhaust from a vehicle. The exhaust is purged from these bags before each and every test. With a one-shot timer, a module could initiate the purge function and then suspend itself for a predetermined period of time while the sample bags are prepared.

  Program schedule requested 20 seconds after call is issued.

```
    R                   S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

  [Time (in Seconds)]

  To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with the TIMER_ABSTIME flag reset, and with an initial expiration value of 20 seconds and a repetition interval of zero.

  Note that if the program wishes merely to suspend itself for the specified interval, it could more easily use *nanosleep*( ).

  — One-shot Timer (Absolute Time): Data Transmission

The results from an experiment are often moved to a different system within a network for postprocessing or archiving. With an absolute one-shot timer, a module that moves data from a test-cell computer to a host computer can be automatically scheduled on a daily basis.

Program schedule requested for 2:30 a.m.

```
         R                                     S
-----+-----+-----+-----+-----+-----+-----+-----+-----+----->
   23:00 23:30 24:00 00:30 01:00 01:30 02:00 02:30 03:00
```

[Time of Day]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with the TIMER_ABSTIME flag set, and an initial expiration value equal to 2:30 a.m. of the next day.

- Periodic Timer (Relative Time): Signal Stabilization

Some measurement devices, such as emission analyzers, do not respond instantaneously to an introduced sample. With a periodic timer with a relative initial expiration time, a module that introduces a sample and records the average response could suspend itself for a predetermined period of time while the signal is stabilized and then sample at a fixed rate.

Program schedule requested 15 seconds after call is issued and every 2 seconds thereafter.

```
   R              S S S S S S S S S S S S S S S S S S S S
----+----+----+----+----+----+----+----+----+----+----+--->
    5   10   15   20   25   30   35   40   45   50   55
```

[Time (in Seconds)]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with TIMER_ABSTIME flag reset, and with an initial expiration value of 15 seconds and a repetition interval of 2 seconds.

- Periodic Timer (Absolute Time): Work Shift-related Processing

Resource utilization data is useful when time to perform experiments is being scheduled at a facility. With a periodic timer with an absolute initial expiration time, a module can be scheduled at the beginning of a work shift to gather resource utilization data throughout the shift. This data can be used to allocate resources effectively to minimize bottlenecks and delays and maximize facility throughput.

Program schedule requested for 2:00 a.m. and every 15 minutes thereafter.

```
         R                               S  S  S  S  S  S
-----+-----+-----+-----+-----+-----+-----+-----+-----+----->
   23:00 23:30 24:00 00:30 01:00 01:30 02:00 02:30 03:00
```

[Time of Day]

To achieve this type of scheduling using the specified facilities, one would allocate a per-process timer based on clock ID CLOCK_REALTIME. Then the timer would be armed via a call to *timer_settime*( ) with TIMER_ABSTIME flag set, and with an initial expiration value equal to 2:00 a.m. and a repetition interval equal to 15 minutes.

- Relationship of Timers to Clocks

  The relationship between clocks and timers armed with an absolute time is straightforward: a timer expiration signal is requested when the associated clock reaches or exceeds the specified time. The relationship between clocks and timers armed with a relative time (an interval) is less obvious, but not unintuitive. In this case, a timer expiration signal is requested when the specified interval, *as measured by the associated clock*, has passed. For the required CLOCK_REALTIME clock, this allows timer expiration signals to be requested at specified ‘‘wall clock’’ times (absolute), or when a specified interval of ‘‘realtime’’ has passed (relative). For an implementation-defined clock—say, a process virtual time clock—timer expirations could be requested when the process has used a specified total amount of virtual time (absolute), or when it has used a specified *additional* amount of virtual time (relative).

  The interfaces also allow flexibility in the implementation of the functions. For example, an implementation could convert all absolute times to intervals by subtracting the clock value at the time of the call from the requested expiration time and ‘‘counting down’’ at the supported resolution. Or it could convert all relative times to absolute expiration time by adding in the clock value at the time of the call and comparing the clock value to the expiration time at the supported resolution. Or it might even choose to maintain absolute times as absolute and compare them to the clock value at the supported resolution for absolute timers, and maintain relative times as intervals and count them down at the resolution supported for relative timers. The choice will be driven by efficiency considerations and the underlying hardware or software clock implementation.

- Data Definitions for Clocks and Timers

  IEEE Std 1003.1-200x uses a time representation capable of supporting nanosecond resolution timers for the following reasons:

  — To enable IEEE Std 1003.1-200x to represent those computer systems already using nanosecond or submicrosecond resolution clocks.

  — To accommodate those per-process timers that might need nanoseconds to specify an absolute value of system-wide clocks, even though the resolution of the per-process timer may only be milliseconds, or *vice versa*.

  — Because the number of nanoseconds in a second can be represented in 32 bits.

  Time values are represented in the **timespec** structure. The *tv_sec* member is of type **time_t** so that this member is compatible with time values used by POSIX.1 functions and the ISO C standard. The *tv_nsec* member is a **signed long** in order to simplify and clarify code that decrements or finds differences of time values. Note that because 1 billion (number of nanoseconds per second) is less than half of the value representable by a signed 32-bit value, it is always possible to add two valid fractional seconds represented as integral nanoseconds without overflowing the signed 32-bit value.

  A maximum allowable resolution for the CLOCK_REALTIME clock of 20 ms (1/50 seconds) was chosen to allow line frequency clocks in European countries to be conforming. 60 Hz clocks in the U.S. will also be conforming, as will finer granularity clocks, although a Strictly Conforming Application cannot assume a granularity of less than 20 ms (1/50 seconds).

  The minimum allowable maximum time allowed for the CLOCK_REALTIME clock and the function *nanosleep*( ), and timers created with *clock_id*=CLOCK_REALTIME, is determined by the fact that the *tv_sec* member is of type **time_t**.

  IEEE Std 1003.1-200x specifies that timer expirations shall not be delivered early, nor shall *nanosleep*( ) return early due to quantization error. IEEE Std 1003.1-200x discusses the various implementations of *alarm*( ) in the rationale and states that implementations that do not allow alarm signals to occur early are the most appropriate, but refrained from mandating this behavior. Because of the importance of predictability to realtime applications, IEEE Std 1003.1-200x takes a stronger stance.

The developers of IEEE Std 1003.1-200x considered using a time representation that differs from POSIX.1b in the second 32 bit of the 64-bit value. Whereas POSIX.1b defines this field as a fractional second in nanoseconds, the other methodology defines this as a binary fraction of one second, with the radix point assumed before the most significant bit.

POSIX.1b is a software, source-level standard and most of the benefits of the alternate representation are enjoyed by hardware implementations of clocks and algorithms. It was felt that mandating this format for POSIX.1b clocks and timers would unnecessarily burden the application writer with writing, possibly non-portable, multiple precision arithmetic packages to perform conversion between binary fractions and integral units such as nanoseconds, milliseconds, and so on.

Rationale for the Monotonic Clock

For those applications that use time services to achieve realtime behavior, changing the value of the clock on which these services rely may cause erroneous timing behavior. For these applications, it is necessary to have a monotonic clock which cannot run backwards, and which has a maximum clock jump that is required to be documented by the implementation. Additionally, it is desirable (but not required by IEEE Std 1003.1-200x) that the monotonic clock increases its value uniformly. This clock should not be affected by changes to the system time; for example, to synchronize the clock with an external source or to account for leap seconds. Such changes would cause errors in the measurement of time intervals for those time services that use the absolute value of the clock.

One could argue that by defining the behavior of time services when the value of a clock is changed, deterministic realtime behavior can be achieved. For example, one could specify that relative time services should be unaffected by changes in the value of a clock. However, there are time services that are based upon an absolute time, but that are essentially intended as relative time services. For example, *pthread_cond_timedwait*( ) uses an absolute time to allow it to wake up after the required interval despite spurious wakeups. Although sometimes the *pthread_cond_timedwait*( ) timeouts are absolute in nature, there are many occasions in which they are relative, and their absolute value is determined from the current time plus a relative time interval. In this latter case, if the clock changes while the thread is waiting, the wait interval will not be the expected length. If a *pthread_cond_timedwait*( ) function were created that would take a relative time, it would not solve the problem because to retain the intended ‘‘deadline’’ a thread would need to compensate for latency due to the spurious wakeup, and preemption between wakeup and the next wait.

The solution is to create a new monotonic clock, whose value does not change except for the regular ticking of the clock, and use this clock for implementing the various relative timeouts that appear in the different POSIX interfaces, as well as allow *pthread_cond_timedwait*( ) to choose this new clock for its timeout. A new *clock_nanosleep*( ) function is created to allow an application to take advantage of this newly defined clock. Notice that the monotonic clock may be implemented using the same hardware clock as the system clock.

Relative timeouts for *sigtimedwait*( ) and *aio_suspend*( ) have been redefined to use the monotonic clock, if present. The *alarm*( ) function has not been redefined, because the same effect but with better resolution can be achieved by creating a timer (for which the appropriate clock may be chosen).

The *pthread_cond_timedwait*( ) function has been treated in a different way, compared to other functions with absolute timeouts, because it is used to wait for an event, and thus it may have a deadline, while the other timeouts are generally used as an error recovery mechanism, and for them the use of the monotonic clock is not so important. Since the desired timeout for the *pthread_cond_timedwait*( ) function may either be a relative interval, or an absolute time of day deadline, a new initialization attribute has been created for condition variables, to specify the clock that shall be used for measuring the timeout in a call to *pthread_cond_timedwait*( ). In this way, if a relative timeout is desired, the monotonic clock will be used; if an absolute deadline is required instead, the CLOCK_REALTIME or another appropriate clock may be used. This capability has not been added to other functions with absolute timeouts because for those functions the expected use of the timeout is mostly to prevent errors, and not so often to meet precise deadlines. As a consequence, the complexity of adding this capability is not justified by its perceived application usage.

The *nanosleep*( ) function has not been modified with the introduction of the monotonic clock. Instead, a new *clock_nanosleep*( ) function has been created, in which the desired clock may be specified in the function call.

- History of Resolution Issues

  Due to the shift from relative to absolute timeouts in IEEE Std 1003.1d-1999, the amendments to the *sem_timedwait*( ), *pthread_mutex_timedlock*( ), *mq_timedreceive*( ), and *mq_timedsend*( ) functions of that standard have been removed. Those amendments specified that CLOCK_MONOTONIC would be used for the (relative) timeouts if the Monotonic Clock option was supported.

  Having these functions continue to be tied solely to CLOCK_MONOTONIC would not work. Since the absolute value of a time value obtained from CLOCK_MONOTONIC is unspecified, under the absolute timeouts interface, applications would behave differently depending on whether the Monotonic Clock option was supported or not (because the absolute value of the clock would have different meanings in either case).

  Two options were considered:

  1. Leave the current behavior unchanged, which specifies the CLOCK_REALTIME clock for these (absolute) timeouts, to allow portability of applications between implementations supporting or not the Monotonic Clock option.

  2. Modify these functions in the way that *pthread_cond_timedwait*( ) was modified to allow a choice of clock, so that an application could use CLOCK_REALTIME when it is trying to achieve an absolute timeout and CLOCK_MONOTONIC when it is trying to achieve a relative timeout.

  It was decided that the features of CLOCK_MONOTONIC are not as critical to these functions as they are to *pthread_cond_timedwait*( ). The *pthread_cond_timedwait*( ) function is given a relative timeout; the timeout may represent a deadline for an event. When these functions are given relative timeouts, the timeouts are typically for error recovery purposes and need not be so precise.

  Therefore, it was decided that these functions should be tied to CLOCK_REALTIME and not complicated by being given a choice of clock.

Execution Time Monitoring

- Introduction

The main goals of the execution time monitoring facilities defined in this chapter are to measure the execution time of processes and threads and to allow an application to establish CPU time limits for these entities.

The analysis phase of time-critical realtime systems often relies on the measurement of execution times of individual threads or processes to determine whether the timing requirements will be met. Also, performance analysis techniques for soft deadline realtime systems rely heavily on the determination of these execution times. The execution time monitoring functions provide application developers with the ability to measure these execution times online and open the possibility of dynamic execution-time analysis and system reconfiguration, if required.

The second goal of allowing an application to establish execution time limits for individual processes or threads and detecting when they overrun allows program robustness to be increased by enabling online checking of the execution times.

If errors are detected—possibly because of erroneous program constructs, the existence of errors in the analysis phase, or a burst of event arrivals—online detection and recovery is possible in a portable way. This feature can be extremely important for many time-critical applications. Other applications require trapping CPU-time errors as a normal way to exit an algorithm; for instance, some realtime artificial intelligence applications trigger a number of independent inference processes of varying accuracy and speed, limit how long they can run, and pick the best answer available when time runs out. In many periodic systems, overrun processes are simply restarted in the next resource period, after necessary end-of-period actions have been taken. This allows algorithms that are inherently data-dependent to be made predictable.

The interface that appears in this chapter defines a new type of clock, the CPU-time clock, which measures execution time. Each process or thread can invoke the clock and timer functions defined in POSIX.1 to use them. Functions are also provided to access the CPU-time clock of other processes or threads to enable remote monitoring of these clocks. Monitoring of threads of other processes is not supported, since these threads are not visible from outside of their own process with the interfaces defined in POSIX.1.

- Execution Time Monitoring Interface

The clock and timer interface defined in POSIX.1 historically only defined one clock, which measures wall-clock time. The requirements for measuring execution time of processes and threads, and setting limits to their execution time by detecting when they overrun, can be accomplished with that interface if a new kind of clock is defined. These new clocks measure execution time, and one is associated with each process and with each thread. The clock functions currently defined in POSIX.1 can be used to read and set these CPU-time clocks, and timers can be created using these clocks as their timing base. These timers can then be used to send a signal when some specified execution time has been exceeded. The CPU-time clocks of each process or thread can be accessed by using the symbols CLOCK_PROCESS_CPUTIME_ID or CLOCK_THREAD_CPUTIME_ID.

The clock and timer interface defined in POSIX.1 and extended with the new kind of CPU-time clock would only allow processes or threads to access their own CPU-time clocks. However, many realtime systems require the possibility of monitoring the execution time of processes or threads from independent monitoring entities. In order to allow applications to construct independent monitoring entities that do not require cooperation from or modification of the monitored entities, two functions have been added: *clock_getcpuclockid*( ), for accessing CPU-time clocks of other processes, and *pthread_getcpuclockid*( ), for accessing CPU-time clocks of other threads. These functions return the clock identifier associated with the process or thread specified in the call. These clock IDs can then be used in the rest of the clock function calls.

The clocks accessed through these functions could also be used as a timing base for the creation of timers, thereby allowing independent monitoring entities to limit the CPU-time consumed by other entities. However, this possibility would imply additional complexity and overhead because of the need to maintain a timer queue for each process or thread, to store the different expiration times associated with timers created by different processes or threads. The working group decided this additional overhead was not justified by application requirements. Therefore, creation of timers attached to the CPU-time clocks of other processes or threads has been specified as implementation-defined.

- Overhead Considerations

  The measurement of execution time may introduce additional overhead in the thread scheduling, because of the need to keep track of the time consumed by each of these entities. In library-level implementations of threads, the efficiency of scheduling could be somehow compromised because of the need to make a kernel call, at each context switch, to read the process CPU-time clock. Consequently, a thread creation attribute called *cpu-clock-requirement* was defined, to allow threads to disconnect their respective CPU-time clocks. However, the Ballot Group considered that this attribute itself introduced some overhead, and that in current implementations it was not worth the effort. Therefore, the attribute was deleted, and thus thread CPU-time clocks are required for all threads if the Thread CPU-Time Clocks option is supported.

- Accuracy of CPU-time Clocks

  The mechanism used to measure the execution time of processes and threads is specified in IEEE Std 1003.1-200x as implementation-defined. The reason for this is that both the underlying hardware and the implementation architecture have a very strong influence on the accuracy achievable for measuring CPU time. For some implementations, the specification of strict accuracy requirements would represent very large overheads, or even the impossibility of being implemented.

  Since the mechanism for measuring execution time is implementation-defined, realtime applications will be able to take advantage of accurate implementations using a portable interface. Of course, strictly conforming applications cannot rely on any particular degree of accuracy, in the same way as they cannot rely on a very accurate measurement of wall clock time. There will always exist applications whose accuracy or efficiency requirements on the implementation are more rigid than the values defined in IEEE Std 1003.1-200x or any other standard.

  In any case, there is a minimum set of characteristics that realtime applications would expect from most implementations. One such characteristic is that the sum of all the execution times of all the threads in a process equals the process execution time, when no CPU-time clocks are disabled. This need not always be the case because implementations may differ in how they account for time during context switches. Another characteristic is that the sum of the execution times of all processes in a system equals the number of processors, multiplied by the elapsed time, assuming that no processor is idle during that elapsed time. However, in some implementations it might not be possible to relate CPU-time to elapsed time. For example, in a heterogeneous multi-processor system in which each processor runs at a different speed, an implementation may choose to define each ‘‘second’’ of CPU-time to be a certain number of ‘‘cycles’’ that a CPU has executed.

- Existing Practice

  Measuring and limiting the execution time of each concurrent activity are common features of most industrial implementations of realtime systems. Almost all critical realtime systems are currently built upon a cyclic executive. With this approach, a regular timer interrupt kicks off the next sequence of computations. It also checks that the current sequence has completed. If it has not, then some error recovery action can be undertaken (or at least an overrun is avoided). Current software engineering principles and the increasing complexity of software are driving application developers to implement these systems on multi-threaded or multi-process operating systems. Therefore, if a POSIX operating system is to be used for this type of application, then it must offer the same level of protection.

  Execution time clocks are also common in most UNIX implementations, although these clocks usually have requirements different from those of realtime applications. The POSIX.1 *times*( ) function supports the measurement of the execution time of the calling process, and its terminated child processes. This execution time is measured in clock ticks and is supplied as two different values with the user and system execution times, respectively. BSD supports the function *getrusage*( ), which allows the calling process to get information about the resources used by itself and/or all of its terminated child processes. The resource usage includes user and system CPU time. Some UNIX systems have options to specify high resolution (up to one microsecond) CPU time clocks using the *times*( ) or the *getrusage*( ) functions.

  The *times*( ) and *getrusage*( ) interfaces do not meet important realtime requirements, such as the possibility of monitoring execution time from a different process or thread, or the possibility of detecting an execution time overrun. The latter requirement is supported in some UNIX implementations that are able to send a signal when the execution time of a process has exceeded some specified value. For example, BSD defines the functions *getitimer*( ) and *setitimer*( ), which can operate either on a realtime clock (wall-clock), or on virtual-time or profile-time clocks which measure CPU time in two different ways. These functions do not support access to the execution time of other processes.

  IBM's MVS operating system supports per-process and per-thread execution time clocks. It also supports limiting the execution time of a given process.

  Given all this existing practice, the working group considered that the POSIX.1 clocks and timers interface was appropriate to meet most of the requirements that realtime applications have for execution time clocks. Functions were added to get the CPU time clock IDs, and to allow/disallow the thread CPU time clocks (in order to preserve the efficiency of some implementations of threads).

- Clock Constants

  The definition of the manifest constants CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID allows processes or threads, respectively, to access their own execution-time clocks. However, given a process or thread, access to its own execution-time clock is also possible if the clock ID of this clock is obtained through a call to *clock_getcpuclockid*( ) or *pthread_getcpuclockid*( ). Therefore, these constants are not necessary and could be deleted to make the interface simpler. Their existence saves one system call in the first access to the CPU-time clock of each process or thread. The working group considered this issue and decided to leave the constants in IEEE Std 1003.1-200x because they are closer to the POSIX.1b use of clock identifiers.

- Library Implementations of Threads

  In library implementations of threads, kernel entities and library threads can coexist. In this case, if the CPU-time clocks are supported, most of the clock and timer functions will need to have two implementations: one in the thread library, and one in the system calls library. The main difference between these two implementations is that the thread library implementation will have to deal with clocks and timers that reside in the thread space, while the kernel implementation will operate on timers and clocks that reside in kernel space. In the library implementation, if the clock ID refers to a clock that resides in the kernel, a kernel call will have to be made. The correct version of the function can be chosen by specifying the appropriate order for the libraries during the link process.

- History of Resolution Issues: Deletion of the *enable* Attribute

In the draft corresponding to the first balloting round, CPU-time clocks had an attribute called *enable*. This attribute was introduced by the working group to allow implementations to avoid the overhead of measuring execution time for those processes or threads for which this measurement was not required. However, the *enable* attribute got several ballot objections. The main reason was that processes are already required to measure execution time by the POSIX.1 *times*( ) function. Consequently, the enable attribute was considered unnecessary, and was deleted from the draft.

Rationale Relating to Timeouts

- Requirements for Timeouts

Realtime systems which must operate reliably over extended periods without human intervention are characteristic in embedded applications such as avionics, machine control, and space exploration, as well as more mundane applications such as cable TV, security systems, and plant automation. A multi-tasking paradigm, in which many independent and/or cooperating software functions relinquish the processor(s) while waiting for a specific stimulus, resource, condition, or operation completion, is very useful in producing well engineered programs for such systems. For such systems to be robust and fault-tolerant, expected occurrences that are unduly delayed or that never occur must be detected so that appropriate recovery actions may be taken. This is difficult if there is no way for a task to regain control of a processor once it has relinquished control (blocked) awaiting an occurrence which, perhaps because of corrupted code, hardware malfunction, or latent software bugs, will not happen when expected. Therefore, the common practice in realtime operating systems is to provide a capability to timeout such blocking services. Although there are several methods to achieve this already defined by POSIX, none are as reliable or efficient as initiating a timeout simultaneously with initiating a blocking service. This is especially critical in hard-realtime embedded systems because the processors typically have little time reserve, and allowed fault recovery times are measured in milliseconds rather than seconds.

The working group largely agreed that such timeouts were necessary and ought to become part of IEEE Std 1003.1-200x, particularly vendors of realtime operating systems whose customers had already expressed a strong need for timeouts. There was some resistance to inclusion of timeouts in IEEE Std 1003.1-200x because the desired effect, fault tolerance, could, in theory, be achieved using existing facilities and alternative software designs, but there was no compelling evidence that realtime system designers would embrace such designs at the sacrifice of performance and/or simplicity.

- Which Services should be Timed Out?

Originally, the working group considered the prospect of providing timeouts on all blocking services, including those currently existing in POSIX.1, POSIX.1b, and POSIX.1c, and future interfaces to be defined by other working groups, as sort of a general policy. This was rather quickly rejected because of the scope of such a change, and the fact that many of those services would not normally be used in a realtime context. More traditional timesharing solutions to timeout would suffice for most of the POSIX.1 interfaces, while others had asynchronous alternatives which, while more complex to utilize, would be adequate for some realtime and all non-realtime applications.

The list of potential candidates for timeouts was narrowed to the following for further consideration:

- POSIX.1b
  - *sem_wait*( )
  - *mq_receive*( )
  - *mq_send*( )
  - *lio_listio*( )
  - *aio_suspend*( )
  - *sigwait*( ) (timeout already implemented by *sigtimedwait*( ))
- POSIX.1c
  - *pthread_mutex_lock*( )
  - *pthread_join*( )
  - *pthread_cond_wait*( ) (timeout already implemented by *pthread_cond_timedwait*( ))
- POSIX.1
  - *read*( )
  - *write*( )

After further review by the working group, the *lio_listio*( ), *read*( ), and *write*( ) functions (all forms of blocking synchronous I/O) were eliminated from the list because of the following:

- Asynchronous alternatives exist
- Timeouts can be implemented, albeit non-portably, in device drivers
- A strong desire not to introduce modifications to POSIX.1 interfaces

The working group ultimately rejected *pthread_join*( ) since both that interface and a timed variant of that interface are non-minimal and may be implemented as a function. See below for a library implementation of *pthread_join*( ).

Thus, there was a consensus among the working group members to add timeouts to 4 of the remaining 5 functions (the timeout for *aio_suspend*( ) was ultimately added directly to POSIX.1b, while the others were added by POSIX.1d). However, *pthread_mutex_lock*( ) remained contentious.

Many feel that *pthread_mutex_lock*( ) falls into the same class as the other functions; that is, it is desirable to timeout a mutex lock because a mutex may fail to be unlocked due to errant or corrupted code in a critical section (looping or branching outside of the unlock code), and therefore is equally in need of a reliable, simple, and efficient timeout. In fact, since mutexes are intended to guard small critical sections, most *pthread_mutex_lock*( ) calls would be expected to obtain the lock without blocking nor utilizing any kernel service, even in implementations of threads with global contention scope; the timeout alternative need only be considered after it is determined that the thread must block.

Those opposed to timing out mutexes feel that the very simplicity of the mutex is compromised by adding a timeout semantic, and that to do so is senseless. They claim that if a timed mutex is really deemed useful by a particular application, then it can be constructed from the facilities already in POSIX.1b and POSIX.1c. The following two C-language library implementations of mutex locking with timeout represent the solutions offered (in both implementations, the timeout parameter is specified as absolute time, not relative time as in the proposed POSIX.1c interfaces).

- Spinlock Implementation

```
#include <pthread.h>
#include <time.h>
#include <errno.h>

int pthread_mutex_timedlock(pthread_mutex_t *mutex,
        const struct timespec *timeout)
    {
    struct timespec timenow;

    while (pthread_mutex_trylock(mutex) == EBUSY)
        {
        clock_gettime(CLOCK_REALTIME, &timenow);
        if (timespec_cmp(&timenow,timeout) >= 0)
            {
            return ETIMEDOUT;
        }
        pthread_yield();
        }
    return 0;
    }
```

The Spinlock implementation is generally unsuitable for any application using priority-based thread scheduling policies such as SCHED_FIFO or SCHED_RR, since the mutex could currently be held by a thread of lower priority within the same allocation domain, but since the waiting thread never blocks, only threads of equal or higher priority will ever run, and the mutex cannot be unlocked. Setting priority inheritance or priority ceiling protocol on the mutex does not solve this problem, since the priority of a mutex owning thread is only boosted if higher priority threads are blocked waiting for the mutex; clearly not the case for this spinlock.

- Condition Wait Implementation

```
#include <pthread.h>
#include <time.h>
#include <errno.h>

struct timed_mutex
    {
    int locked;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    };
typedef struct timed_mutex timed_mutex_t;

int timed_mutex_lock(timed_mutex_t *tm,
        const struct timespec *timeout)
    {
    int timedout=FALSE;
    int error_status;
```

```
    pthread_mutex_lock(&tm->mutex);
    while (tm->locked && !timedout)
        {
        if ((error_status=pthread_cond_timedwait(&tm->cond,
            &tm->mutex,
            timeout))!=0)
        {
        if (error_status==ETIMEDOUT) timedout = TRUE;
        }
    }
    if(timedout)
        {
        pthread_mutex_unlock(&tm->mutex);
        return ETIMEDOUT;
        }
    else
        {
        tm->locked = TRUE;
        pthread_mutex_unlock(&tm->mutex);
        return 0;
        }
    }
void timed_mutex_unlock(timed_mutex_t *tm)
    {
    pthread_mutex_lock(&tm->mutex); / for case assignment not atomic /
    tm->locked = FALSE;
    pthread_mutex_unlock(&tm->mutex);
    pthread_cond_signal(&tm->cond);
    }
```

The Condition Wait implementation effectively substitutes the *pthread_cond_timedwait*( ) function (which is currently timed out) for the desired *pthread_mutex_timedlock*( ). Since waits on condition variables currently do not include protocols which avoid priority inversion, this method is generally unsuitable for realtime applications because it does not provide the same priority inversion protection as the untimed *pthread_mutex_lock*( ). Also, for any given implementations of the current mutex and condition variable primitives, this library implementation has a performance cost at least 2.5 times that of the untimed *pthread_mutex_lock*( ) even in the case where the timed mutex is readily locked without blocking (the interfaces required for this case are shown in bold). Even in uniprocessors or where assignment is atomic, at least an additional *pthread_cond_signal*( ) is required. *pthread_mutex_timedlock*( ) could be implemented at effectively no performance penalty in this case because the timeout parameters need only be considered after it is determined that the mutex cannot be locked immediately.

Thus it has not yet been shown that the full semantics of mutex locking with timeout can be efficiently and reliably achieved using existing interfaces. Even if the existence of an acceptable library implementation were proven, it is difficult to justify why the interface itself should not be made portable, especially considering approval for the other four timeouts.

- Rationale for Library Implementation of *pthread_timedjoin*( )

Library implementation of *pthread_timedjoin*( ):

```
/*
 * Construct a thread variety entirely from existing functions
 * with which a join can be done, allowing the join to time out.
 */
#include <pthread.h>
#include <time.h>

struct timed_thread {
    pthread_t t;
    pthread_mutex_t m;
    int exiting;
    pthread_cond_t exit_c;
    void *(*start_routine)(void *arg);
    void *arg;
    void *status;
};

typedef struct timed_thread *timed_thread_t;
static pthread_key_t timed_thread_key;
static pthread_once_t timed_thread_once = PTHREAD_ONCE_INIT;

static void timed_thread_init()
{
    pthread_key_create(&timed_thread_key, NULL);
}

static void *timed_thread_start_routine(void *args)

/*
 * Routine to establish thread-specific data value and run the actual
 * thread start routine which was supplied to timed_thread_create().
 */
{
    timed_thread_t tt = (timed_thread_t) args;

    pthread_once(&timed_thread_once, timed_thread_init);
    pthread_setspecific(timed_thread_key, (void *)tt);
    timed_thread_exit((tt->start_routine)(tt->arg));
}

int timed_thread_create(timed_thread_t ttp, const pthread_attr_t *attr,
    void *(*start_routine)(void *), void *arg)

/*
 * Allocate a thread which can be used with timed_thread_join().
 */
{
    timed_thread_t tt;
    int result;

    tt = (timed_thread_t) malloc(sizeof(struct timed_thread));
    pthread_mutex_init(&tt->m,NULL);
    tt->exiting = FALSE;
    pthread_cond_init(&tt->exit_c,NULL);
    tt->start_routine = start_routine;
```

```
        tt->arg = arg;
        tt->status = NULL;

        if ((result = pthread_create(&tt->t, attr,
            timed_thread_start_routine, (void *)tt)) != 0) {
            free(tt);
            return result;
        }

        pthread_detach(tt->t);
        ttp = tt;
        return 0;
    }

    int timed_thread_join(timed_thread_t tt,
        struct timespec *timeout,
        void **status)
    {
        int result;

        pthread_mutex_lock(&tt->m);
        result = 0;
        /*
         * Wait until the thread announces that it is exiting,
         * or until timeout.
         */
        while (result == 0 && ! tt->exiting) {
            result = pthread_cond_timedwait(&tt->exit_c, &tt->m, timeout);
        }
        pthread_mutex_unlock(&tt->m);
        if (result == 0 && tt->exiting) {
            *status = tt->status;
            free((void *)tt);
            return result;
        }
        return result;
    }

    void timed_thread_exit(void *status)
    {
        timed_thread_t tt;
        void *specific;

        if ((specific=pthread_getspecific(timed_thread_key)) == NULL){
            /*
             * Handle cases which won’t happen with correct usage.
             */
            pthread_exit( NULL);
        }
        tt = (timed_thread_t) specific;
        pthread_mutex_lock(&tt->m);
        /*
         * Tell a joiner that we’re exiting.
         */
        tt->status = status;
```

```
        tt->exiting = TRUE;
        pthread_cond_signal(&tt->exit_c);
        pthread_mutex_unlock(&tt->m);
        /*
         * Call pthread exit() to call destructors and really
         * exit the thread.
         */
        pthread_exit(NULL);
}
```

The *pthread_join*( ) C-language example shown above demonstrates that it is possible, using existing pthread facilities, to construct a variety of thread which allows for joining such a thread, but which allows the join operation to time out. It does this by using a *pthread_cond_timedwait*( ) to wait for the thread to exit. A **timed_thread_t** descriptor structure is used to pass parameters from the creating thread to the created thread, and from the exiting thread to the joining thread. This implementation is roughly equivalent to what a normal *pthread_join*( ) implementation would do, with the single change being that *pthread_cond_timedwait*( ) is used in place of a simple *pthread_cond_wait*( ).

Since it is possible to implement such a facility entirely from existing pthread interfaces, and with roughly equal efficiency and complexity to an implementation which would be provided directly by a pthreads implementation, it was the consensus of the working group members that any *pthread_timedjoin*( ) facility would be unnecessary, and should not be provided.

- Form of the Timeout Interfaces

The working group considered a number of alternative ways to add timeouts to blocking services. At first, a system interface which would specify a one-shot or persistent timeout to be applied to subsequent blocking services invoked by the calling process or thread was considered because it allowed all blocking services to be timed out in a uniform manner with a single additional interface; this was rather quickly rejected because it could easily result in the wrong services being timed out.

It was suggested that a timeout value might be specified as an attribute of the object (semaphore, mutex, message queue, and so on), but there was no consensus on this, either on a case-by-case basis or for all timeouts.

Looking at the two existing timeouts for blocking services indicates that the working group members favor a separate interface for the timed version of a function. However, *pthread_cond_timedwait*( ) utilizes an absolute timeout value while *sigtimedwait*( ) uses a relative timeout value. The working group members agreed that relative timeout values are appropriate where the timeout mechanism's primary use was to deal with an unexpected or error situation, but they are inappropriate when the timeout must expire at a particular time, or before a specific deadline. For the timeouts being introduced in IEEE Std 1003.1-200x, the working group considered allowing both relative and absolute timeouts as is done with POSIX.1b timers, but ultimately favored the simpler absolute timeout form.

An absolute time measure can be easily implemented on top of an interface that specifies relative time, by reading the clock, calculating the difference between the current time and the desired wake-up time, and issuing a relative timeout call. But there is a race condition with this approach because the thread could be preempted after reading the clock, but before making the timed out call; in this case, the thread would be awakened later than it should and, thus, if the wake up time represented a deadline, it would miss it.

There is also a race condition when trying to build a relative timeout on top of an interface that specifies absolute timeouts. In this case, we would have to read the clock to calculate the absolute wake-up time as the sum of the current time plus the relative timeout interval. In this case, if the thread is preempted after reading the clock but before making the timed out call, the thread would be awakened earlier than desired.

But the race condition with the absolute timeouts interface is not as bad as the one that happens with the relative timeout interface, because there are simple workarounds. For the absolute timeouts interface, if the timing requirement is a deadline, we can still meet this deadline because the thread woke up earlier than the deadline. If the timeout is just used as an error recovery mechanism, the precision of timing is not really important. If the timing requirement is that between actions A and B a minimum interval of time must elapse, we can safely use the absolute timeout interface by reading the clock after action A has been started. It could be argued that, since the call with the absolute timeout is atomic from the application point of view, it is not possible to read the clock after action A, if this action is part of the timed out call. But if we look at the nature of the calls for which we specify timeouts (locking a mutex, waiting for a semaphore, waiting for a message, or waiting until there is space in a message queue), the timeouts that an application would build on these actions would not be triggered by these actions themselves, but by some other external action. For example, if we want to wait for a message to arrive to a message queue, and wait for at least 20 milliseconds, this time interval would start to be counted from some event that would trigger both the action that produces the message, as well as the action that waits for the message to arrive, and not by the wait-for-message operation itself. In this case, we could use the workaround proposed above.

For these reasons, the absolute timeout is preferred over the relative timeout interface.

B.2.9 Threads

Threads will normally be more expensive than subroutines (or functions, routines, and so on) if specialized hardware support is not provided. Nevertheless, threads should be sufficiently efficient to encourage their use as a medium to fine-grained structuring mechanism for parallelism in an application. Structuring an application using threads then allows it to take immediate advantage of any underlying parallelism available in the host environment. This means implementors are encouraged to optimize for fast execution at the possible expense of efficient utilization of storage. For example, a common thread creation technique is to cache appropriate thread data structures. That is, rather than releasing system resources, the implementation retains these resources and reuses them when the program next asks to create a new thread. If this reuse of thread resources is to be possible, there has to be very little unique state associated with each thread, because any such state has to be reset when the thread is reused.

Thread Creation Attributes

Attributes objects are provided for threads, mutexes, and condition variables as a mechanism to support probable future standardization in these areas without requiring that the interface itself be changed. Attributes objects provide clean isolation of the configurable aspects of threads. For example, ‘‘stack size’’ is an important attribute of a thread, but it cannot be expressed portably. When porting a threaded program, stack sizes often need to be adjusted. The use of attributes objects can help by allowing the changes to be isolated in a single place, rather than being spread across every instance of thread creation.

Attributes objects can be used to set up *classes* of threads with similar attributes; for example, ‘‘threads with large stacks and high priority’’ or ‘‘threads with minimal stacks’’. These classes can be defined in a single place and then referenced wherever threads need to be created.

Changes to ‘‘class’’ decisions become straightforward, and detailed analysis of each *pthread_create*( ) call is not required.

The attributes objects are defined as opaque types as an aid to extensibility. If these objects had been specified as structures, adding new attributes would force recompilation of all multi-threaded programs when the attributes objects are extended; this might not be possible if different program components were supplied by different vendors.

Additionally, opaque attributes objects present opportunities for improving performance. Argument validity can be checked once when attributes are set, rather than each time a thread is created. Implementations will often need to cache kernel objects that are expensive to create. Opaque attributes objects provide an efficient mechanism to detect when cached objects become invalid due to attribute changes.

Because assignment is not necessarily defined on a given opaque type, implementation-dependent default values cannot be defined in a portable way. The solution to this problem is to allow attribute objects to be initialized dynamically by attributes object initialization functions, so that default values can be supplied automatically by the implementation.

The following proposal was provided as a suggested alternative to the supplied attributes:

1. Maintain the style of passing a parameter formed by the bitwise-inclusive OR of flags to the initialization routines (*pthread_create*( ), *pthread_mutex_init*( ), *pthread_cond_init*( )). The parameter containing the flags should be an opaque type for extensibility. If no flags are set in the parameter, then the objects are created with default characteristics. An implementation may specify implementation-defined flag values and associated behavior.
2. If further specialization of mutexes and condition variables is necessary, implementations may specify additional procedures that operate on the **pthread_mutex_t** and **pthread_cond_t** objects (instead of on attributes objects).

The difficulties with this solution are:

1. A bitmask is not opaque if bits have to be set into bit-vector attributes objects using explicitly-coded bitwise-inclusive OR operations. If the set of options exceeds an **int**, application programmers need to know the location of each bit. If bits are set or read by encapsulation (that is, *get**( ) or *set**( ) functions), then the bitmask is merely an implementation of attributes objects as currently defined and should not be exposed to the programmer.
2. Many attributes are not Boolean or very small integral values. For example, scheduling policy may be placed in 3 bits or 4 bits, but priority requires 5 bits or more, thereby taking up at least 8 bits out of a possible 16 bits on machines with 16-bit integers. Because of this, the bitmask can only reasonably control whether particular attributes are set or not, and it cannot serve as the repository of the value itself. The value needs to be specified as a function parameter (which is non-extensible), or by setting a structure field (which is non-opaque), or by *get**( ) and *set**( ) functions (making the bitmask a redundant addition to the attributes objects).

Stack size is defined as an optional attribute because the very notion of a stack is inherently machine-dependent. Some implementations may not be able to change the size of the stack, for example, and others may not need to because stack pages may be discontiguous and can be allocated and released on demand.

The attribute mechanism has been designed in large measure for extensibility. Future extensions to the attribute mechanism or to any attributes object defined in IEEE Std 1003.1-200x has to be done with care so as not to affect binary-compatibility.

Attribute objects, even if allocated by means of dynamic allocation functions such as *malloc*( ), may have their size fixed at compile time. This means, for example, a *pthread_create*( ) in an implementation with extensions to the **pthread_attr_t** cannot look beyond the area that the binary application assumes is valid. This suggests that implementations should maintain a size field in the attributes object, as well as possibly version information, if extensions in different directions (possibly by different vendors) are to be accommodated.

Thread Implementation Models

There are various thread implementation models. At one end of the spectrum is the ''library-thread model''. In such a model, the threads of a process are not visible to the operating system kernel, and the threads are not kernel scheduled entities. The process is the only kernel scheduled entity. The process is scheduled onto the processor by the kernel according to the scheduling attributes of the process. The threads are scheduled onto the single kernel scheduled entity (the process) by the runtime library according to the scheduling attributes of the threads. A problem with this model is that it constrains concurrency. Since there is only one kernel scheduled entity (namely, the process), only one thread per process can execute at a time. If the thread that is executing blocks on I/O, then the whole process blocks.

At the other end of the spectrum is the ''kernel-thread model''. In this model, all threads are visible to the operating system kernel. Thus, all threads are kernel scheduled entities, and all threads can concurrently execute. The threads are scheduled onto processors by the kernel according to the scheduling attributes of the threads. The drawback to this model is that the creation and management of the threads entails operating system calls, as opposed to subroutine calls, which makes kernel threads heavier weight than library threads.

Hybrids of these two models are common. A hybrid model offers the speed of library threads and the concurrency of kernel threads. In hybrid models, a process has some (relatively small) number of kernel scheduled entities associated with it. It also has a potentially much larger number of library threads associated with it. Some library threads may be bound to kernel scheduled entities, while the other library threads are multiplexed onto the remaining kernel scheduled entities. There are two levels of thread scheduling:

1. The runtime library manages the scheduling of (unbound) library threads onto kernel scheduled entities.

2. The kernel manages the scheduling of kernel scheduled entities onto processors.

For this reason, a hybrid model is referred to as a *two-level threads scheduling model*. In this model, the process can have multiple concurrently executing threads; specifically, it can have as many concurrently executing threads as it has kernel scheduled entities.

Thread-Specific Data

Many applications require that a certain amount of context be maintained on a per-thread basis across procedure calls. A common example is a multi-threaded library routine that allocates resources from a common pool and maintains an active resource list for each thread. The thread-specific data interface provided to meet these needs may be viewed as a two-dimensional array of values with keys serving as the row index and thread IDs as the column index (although the implementation need not work this way).

- Models

  Three possible thread-specific data models were considered:

  1. No Explicit Support

A standard thread-specific data interface is not strictly necessary to support applications that require per-thread context. One could, for example, provide a hash function that converted a **pthread_t** into an integer value that could then be used to index into a global array of per-thread data pointers. This hash function, in conjunction with *pthread_self*( ), would be all the interface required to support a mechanism of this sort. Unfortunately, this technique is cumbersome. It can lead to duplicated code as each set of cooperating modules implements their own per-thread data management schemes.

2. Single (**void** *) Pointer

   Another technique would be to provide a single word of per-thread storage and a pair of functions to fetch and store the value of this word. The word could then hold a pointer to a block of per-thread memory. The allocation, partitioning, and general use of this memory would be entirely up to the application. Although this method is not as problematic as technique 1, it suffers from interoperability problems. For example, all modules using the per-thread pointer would have to agree on a common usage protocol.

3. Key / Value Mechanism

   This method associates an opaque key (for example, stored in a variable of type **pthread_key_t**) with each per-thread datum. These keys play the role of identifiers for per-thread data. This technique is the most generic and avoids the problems noted above, albeit at the cost of some complexity.

The primary advantage of the third model is its information hiding properties. Modules using this model are free to create and use their own key(s) independent of all other such usage, whereas the other models require that all modules that use thread-specific context explicitly cooperate with all other such modules. The data-independence provided by the third model is worth the additional interface.

- Requirements

It is important that it be possible to implement the thread-specific data interface without the use of thread private memory. To do otherwise would increase the weight of each thread, thereby limiting the range of applications for which the threads interfaces provided by IEEE Std 1003.1-200x is appropriate.

The values that one binds to the key via *pthread_setspecific*( ) may, in fact, be pointers to shared storage locations available to all threads. It is only the key / value bindings that are maintained on a per-thread basis, and these can be kept in any portion of the address space that is reserved for use by the calling thread (for example, on the stack). Thus, no per-thread MMU state is required to implement the interface. On the other hand, there is nothing in the interface specification to preclude the use of a per-thread MMU state if it is available (for example, the key values returned by *pthread_key_create*( ) could be thread private memory addresses).

- Standardization Issues

Thread-specific data is a requirement for a usable thread interface. The binding described in this section provides a portable thread-specific data mechanism for languages that do not directly support a thread-specific storage class. A binding to IEEE Std 1003.1-200x for a language that does include such a storage class need not provide this specific interface.

If a language were to include the notion of thread-specific storage, it would be desirable (but *not* required) to provide an implementation of the pthreads thread-specific data interface based on the language feature. For example, assume that a compiler for a C-like language supports a *private* storage class that provides thread-specific storage. Something similar to the following macros might be used to effect a compatible implementation:

```
#define pthread_key_t                      private void *
#define pthread_key_create(key)            /* no-op */
#define pthread_setspecific(key,value)     (key)=(value)
#define pthread_getspecific(key)           (key)
```

Note: For the sake of clarity, this example ignores destructor functions. A correct implementation would have to support them.

Barriers

- Background

Barriers are typically used in parallel DO/FOR loops to ensure that all threads have reached a particular stage in a parallel computation before allowing any to proceed to the next stage. Highly efficient implementation is possible on machines which support a ‘‘Fetch and Add’’ operation as described in the referenced Almasi and Gottlieb (1989).

The use of return value PTHREAD_BARRIER_SERIAL_THREAD is shown in the following example:

```
if ( (status=pthread_barrier_wait(&barrier)) ==
    PTHREAD_BARRIER_SERIAL_THREAD) {
    ...serial section
    }
        else if (status != 0) {
        ...error processing
    }
status=pthread_barrier_wait(&barrier);
...
```

This behavior allows a serial section of code to be executed by one thread as soon as all threads reach the first barrier. The second barrier prevents the other threads from proceeding until the serial section being executed by the one thread has completed.

Although barriers can be implemented with mutexes and condition variables, the referenced Almasi and Gottlieb (1989) provides ample illustration that such implementations are significantly less efficient than is possible. While the relative efficiency of barriers may well vary by implementation, it is important that they be recognized in the IEEE Std 1003.1-200x to facilitate application portability while providing the necessary freedom to implementors.

- Lack of Timeout Feature

Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for barrier waits for the following reasons:

- Multiple threads may use different timeout values, some of which may be indefinite. It is not clear which threads should break through the barrier with a timeout error if and when these timeouts expire.

- The barrier may become unusable once a thread breaks out of a *pthread_barrier_wait*( ) with a timeout error. There is, in general, no way to guarantee the consistency of a barrier’s internal data structures once a thread has timed out of a *pthread_barrier_wait*( ). Even the inclusion of a special barrier reinitialization function would not help much since it is not clear how this function would affect the behavior of threads that reach the barrier between the original timeout and the call to the reinitialization function.

Spin Locks

- Background

  Spin locks represent an extremely low-level synchronization mechanism suitable primarily for use on shared memory multi-processors. It is typically an atomically modified Boolean value that is set to one when the lock is held and to zero when the lock is freed.

  When a caller requests a spin lock that is already held, it typically spins in a loop testing whether the lock has become available. Such spinning wastes processor cycles so the lock should only be held for short durations and not across sleep / block operations. Callers should unlock spin locks before calling sleep operations.

  Spin locks are available on a variety of systems. The functions included in IEEE Std 1003.1-200x are an attempt to standardize that existing practice.

- Lack of Timeout Feature

  Alternate versions of most blocking routines have been provided to support watchdog timeouts. No alternate interface of this sort has been provided for spin locks for the following reasons:

  - It is impossible to determine appropriate timeout intervals for spin locks in a portable manner. The amount of time one can expect to spend spin-waiting is inversely proportional to the degree of parallelism provided by the system.

    It can vary from a few cycles when each competing thread is running on its own processor, to an indefinite amount of time when all threads are multiplexed on a single processor (which is why spin locking is not advisable on uniprocessors).

  - When used properly, the amount of time the calling thread spends waiting on a spin lock should be considerably less than the time required to set up a corresponding watchdog timer. Since the primary purpose of spin locks is to provide a low-overhead synchronization mechanism for multi-processors, the overhead of a timeout mechanism was deemed unacceptable.

  It was also suggested that an additional *count* argument be provided (on the *pthread_spin_lock*( ) call) in *lieu* of a true timeout so that a spin lock call could fail gracefully if it was unable to apply the lock after *count* attempts. This idea was rejected because it is not existing practice. Furthermore, the same effect can be obtained with *pthread_spin_trylock*( ), as illustrated below:

```
int n = MAX_SPIN;

while ( −−n >= 0 )
{
    if ( !pthread_spin_try_lock(...) )
        break;
}
if ( n >= 0 )
{
    /* Successfully acquired the lock */
}
else
{
    /* Unable to acquire the lock */
}
```

- *process-shared* Attribute

  The initialization functions associated with most POSIX synchronization objects (for example, mutexes, barriers, and read-write locks) take an attributes object with a *process-shared* attribute that specifies whether or not the object is to be shared across processes. In the draft corresponding to the first balloting round, two separate initialization functions are provided for spin locks, however: one for spin locks that were to be shared across processes (*spin_init*( )), and one for locks that were only used by multiple threads within a single process (*pthread_spin_init*( )). This was done so as to keep the overhead associated with spin waiting to an absolute minimum. However, the balloting group requested that, since the overhead associated to a bit check was small, spin locks should be consistent with the rest of the synchronization primitives, and thus the *process-shared* attribute was introduced for spin locks.

- Spin Locks *versus* Mutexes

  It has been suggested that mutexes are an adequate synchronization mechanism and spin locks are not necessary. Locking mechanisms typically must trade off the processor resources consumed while setting up to block the thread and the processor resources consumed by the thread while it is blocked. Spin locks require very little resources to set up the blocking of a thread. Existing practice is to simply loop, repeating the atomic locking operation until the lock is available. While the resources consumed to set up blocking of the thread are low, the thread continues to consume processor resources while it is waiting.

  On the other hand, mutexes may be implemented such that the processor resources consumed to block the thread are large relative to a spin lock. After detecting that the mutex lock is not available, the thread must alter its scheduling state, add itself to a set of waiting threads, and, when the lock becomes available again, undo all of this before taking over ownership of the mutex. However, while a thread is blocked by a mutex, no processor resources are consumed.

  Therefore, spin locks and mutexes may be implemented to have different characteristics. Spin locks may have lower overall overhead for very short-term blocking, and mutexes may have lower overall overhead when a thread will be blocked for longer periods of time. The presence of both interfaces allows implementations with these two different characteristics, both of which may be useful to a particular application.

  It has also been suggested that applications can build their own spin locks from the *pthread_mutex_trylock*( ) function:

```
    while (pthread_mutex_trylock(&mutex));
```

The apparent simplicity of this construct is somewhat deceiving, however. While the actual wait is quite efficient, various guarantees on the integrity of mutex objects (for example, priority inheritance rules) may add overhead to the successful path of the trylock operation that is not required of spin locks. One could, of course, add an attribute to the mutex to bypass such overhead, but the very act of finding and testing this attribute represents more overhead than is found in the typical spin lock.

The need to hold spin lock overhead to an absolute minimum also makes it impossible to provide guarantees against starvation similar to those provided for mutexes or read-write locks. The overhead required to implement such guarantees (for example, disabling preemption before spinning) may well exceed the overhead of the spin wait itself by many orders of magnitude. If a ‘‘safe’’ spin wait seems desirable, it can always be provided (albeit at some performance cost) via appropriate mutex attributes.

XSI Supported Functions

On XSI-conformant systems, the following symbolic constants are always defined:

_POSIX_READER_WRITER_LOCKS
_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_PROCESS_SHARED
_POSIX_THREADS

Therefore, the following threads functions are always supported:

pthread_atfork( )
pthread_attr_destroy( )
pthread_attr_getdetachstate( )
pthread_attr_getguardsize( )
pthread_attr_getschedparam( )
pthread_attr_getstack( )
pthread_attr_getstackaddr( )
pthread_attr_getstacksize( )
pthread_attr_init( )
pthread_attr_setdetachstate( )
pthread_attr_setguardsize( )
pthread_attr_setschedparam( )
pthread_attr_setstack( )
pthread_attr_setstackaddr( )
pthread_attr_setstacksize( )
pthread_cancel( )
pthread_cleanup_pop( )
pthread_cleanup_push( )
pthread_cond_broadcast( )
pthread_cond_destroy( )
pthread_cond_init( )
pthread_cond_signal( )
pthread_cond_timedwait( )
pthread_cond_wait( )
pthread_condattr_destroy( )
pthread_key_create( )
pthread_key_delete( )
pthread_kill( )
pthread_mutex_destroy( )
pthread_mutex_init( )
pthread_mutex_lock( )
pthread_mutex_trylock( )
pthread_mutex_unlock( )
pthread_mutexattr_destroy( )
pthread_mutexattr_getpshared( )
pthread_mutexattr_gettype( )
pthread_mutexattr_init( )
pthread_mutexattr_setpshared( )
pthread_mutexattr_settype( )
pthread_once( )
pthread_rwlock_destroy( )
pthread_rwlock_init( )
pthread_rwlock_rdlock( )
pthread_rwlock_tryrdlock( )
pthread_rwlock_trywrlock( )
pthread_rwlock_unlock( )
pthread_rwlock_wrlock( )
pthread_rwlockattr_destroy( )
pthread_rwlockattr_getpshared( )
pthread_rwlockattr_init( )

pthread_condattr_getpshared( )
pthread_condattr_init( )
pthread_condattr_setpshared( )
pthread_create( )
pthread_detach( )
pthread_equal( )
pthread_exit( )
pthread_getconcurrency( )
pthread_getspecific( )
pthread_join( )
pthread_rwlockattr_setpshared( )
pthread_self( )
pthread_setcancelstate( )
pthread_setcanceltype( )
pthread_setconcurrency( )
pthread_setspecific( )
pthread_sigmask( )
pthread_testcancel( )
sigwait( )

On XSI-conformant systems, the symbolic constant _POSIX_THREAD_SAFE_FUNCTIONS is always defined. Therefore, the following functions are always supported:

asctime_r( )
ctime_r( )
flockfile( )
ftrylockfile( )
funlockfile( )
getc_unlocked( )
getchar_unlocked( )
getgrgid_r( )
getgrnam_r( )
getpwnam_r( )
getpwuid_r( )
gmtime_r( )
localtime_r( )
putc_unlocked( )
putchar_unlocked( )
rand_r( )
readdir_r( )
strerror_r( )
strtok_r( )

The following threads functions are only supported on XSI-conformant systems if the Realtime Threads Option Group is supported :

pthread_attr_getinheritsched( )
pthread_attr_getschedpolicy( )
pthread_attr_getscope( )
pthread_attr_setinheritsched( )
pthread_attr_setschedpolicy( )
pthread_attr_setscope( )
pthread_getschedparam( )
pthread_mutex_getprioceiling( )
pthread_mutex_setprioceiling( )
pthread_mutexattr_getprioceiling( )
pthread_mutexattr_getprotocol( )
pthread_mutexattr_setprioceiling( )
pthread_mutexattr_setprotocol( )
pthread_setschedparam( )

XSI Threads Extensions

The following XSI extensions to POSIX.1c are now supported in IEEE Std 1003.1-200x as part of the alignment with the Single UNIX Specification:

- Extended mutex attribute types
- Read-write locks and attributes (also introduced by IEEE Std 1003.1j-2000 amendment)
- Thread concurrency level
- Thread stack guard size
- Parallel I/O

A total of 19 new functions were added.

These extensions carefully follow the threads programming model specified in POSIX.1c. As with POSIX.1c, all the new functions return zero if successful; otherwise, an error number is returned to indicate the error.

The concept of attribute objects was introduced in POSIX.1c to allow implementations to extend IEEE Std 1003.1-200x without changing the existing interfaces. Attribute objects were defined for threads, mutexes, and condition variables. Attributes objects are defined as implementation-defined opaque types to aid extensibility, and functions are defined to allow attributes to be set or retrieved. This model has been followed when adding the new type attribute of **pthread_mutexattr_t** or the new read-write lock attributes object **pthread_rwlockattr_t**.

- Extended Mutex Attributes

  POSIX.1c defines a mutex attributes object as an implementation-defined opaque object of type **pthread_mutexattr_t**, and specifies a number of attributes which this object must have and a number of functions which manipulate these attributes. These attributes include *detachstate*, *inheritsched*, *schedparm*, *schedpolicy*, *contentionscope*, *stackaddr*, and *stacksize*.

  The System Interfaces volume of IEEE Std 1003.1-200x specifies another mutex attribute called *type*. The *type* attribute allows applications to specify the behavior of mutex locking operations in situations where the POSIX.1c behavior is undefined. The OSF DCE threads implementation, based on Draft 4 of POSIX.1c, specified a similar attribute. Note that the names of the attributes have changed somewhat from the OSF DCE threads implementation.

  The System Interfaces volume of IEEE Std 1003.1-200x also extends the specification of the following POSIX.1c functions which manipulate mutexes:

  pthread_mutex_lock( )
  pthread_mutex_trylock( )
  pthread_mutex_unlock( )

  to take account of the new mutex attribute type and to specify behavior which was declared as undefined in POSIX.1c. How a calling thread acquires or releases a mutex now depends upon the mutex *type* attribute.

  The *type* attribute can have the following values:

  PTHREAD_MUTEX_NORMAL
  Basic mutex with no specific error checking built in. Does not report a deadlock error.

  PTHREAD_MUTEX_RECURSIVE
  Allows any thread to recursively lock a mutex. The mutex must be unlocked an equal number of times to release the mutex.

  PTHREAD_MUTEX_ERRORCHECK
  Detects and reports simple usage errors; that is, an attempt to unlock a mutex that is not locked by the calling thread or that is not locked at all, or an attempt to relock a mutex the thread already owns.

  PTHREAD_MUTEX_DEFAULT
  The default mutex type. May be mapped to any of the above mutex types or may be an implementation-defined type.

  Normal mutexes do not detect deadlock conditions; for example, a thread will hang if it tries to relock a normal mutex that it already owns. Attempting to unlock a mutex locked by another thread, or unlocking an unlocked mutex, results in undefined behavior. Normal mutexes will usually be the fastest type of mutex available on a platform but provide the least error checking.

  Recursive mutexes are useful for converting old code where it is difficult to establish clear boundaries of synchronization. A thread can relock a recursive mutex without first unlocking it. The relocking deadlock which can occur with normal mutexes cannot occur with this type of mutex. However, multiple locks of a recursive mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. Furthermore, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock a recursive mutex which another thread has locked returns with an error. A thread attempting to unlock a recursive mutex that is not locked shall return with an error. Never use a recursive mutex with condition variables because the implicit unlock performed by *pthread_cond_wait*( ) or *pthread_cond_timedwait*( ) will not actually release the mutex if it had been locked multiple times.

Errorcheck mutexes provide error checking and are useful primarily as a debugging aid. A thread attempting to relock an errorcheck mutex without first unlocking it returns with an error. Again, this type of mutex maintains the concept of an owner. Thus, a thread attempting to unlock an errorcheck mutex which another thread has locked returns with an error. A thread attempting to unlock an errorcheck mutex that is not locked also returns with an error. It should be noted that errorcheck mutexes will almost always be much slower than normal mutexes due to the extra state checks performed.

The *default* mutex type provides implementation-defined error checking. The default mutex may be mapped to one of the other defined types or may be something entirely different. This enables each vendor to provide the mutex semantics which the vendor feels will be most useful to their target users. Most vendors will probably choose to make normal mutexes the default so as to give applications the benefit of the fastest type of mutexes available on their platform. Check your implementation's documentation.

An application developer can use any of the mutex types almost interchangeably as long as the application does not depend upon the implementation detecting (or failing to detect) any particular errors. Note that a recursive mutex can be used with condition variable waits as long as the application never recursively locks the mutex.

Two functions are provided for manipulating the *type* attribute of a mutex attributes object. This attribute is set or returned in the *type* parameter of these functions. The *pthread_mutexattr_settype*( ) function is used to set a specific type value while *pthread_mutexattr_gettype*( ) is used to return the type of the mutex. Setting the *type* attribute of a mutex attributes object affects only mutexes initialized using that mutex attributes object. Changing the *type* attribute does not affect mutexes previously initialized using that mutex attributes object.

- Read-Write Locks and Attributes

  The read-write locks introduced have been harmonized with those in IEEE Std 1003.1j-2000; see also Section B.2.9.6 (on page 3464).

  Read-write locks (also known as reader-writer locks) allow a thread to exclusively lock some shared data while updating that data, or allow any number of threads to have simultaneous read-only access to the data.

  Unlike a mutex, a read-write lock distinguishes between reading data and writing data. A mutex excludes all other threads. A read-write lock allows other threads access to the data, providing no thread is modifying the data. Thus, a read-write lock is less primitive than either a mutex-condition variable pair or a semaphore.

  Application developers should consider using a read-write lock rather than a mutex to protect data that is frequently referenced but seldom modified. Most threads (readers) will be able to read the data without waiting and will only have to block when some other thread (a writer) is in the process of modifying the data. Conversely a thread that wants to change the data is forced to wait until there are no readers. This type of lock is often used to facilitate parallel access to data on multi-processor platforms or to avoid context switches on single processor platforms where multiple threads access the same data.

If a read-write lock becomes unlocked and there are multiple threads waiting to acquire the write lock, the implementation's scheduling policy determines which thread shall acquire the read-write lock for writing. If there are multiple threads blocked on a read-write lock for both read locks and write locks, it is unspecified whether the readers or a writer acquire the lock first. However, for performance reasons, implementations often favor writers over readers to avoid potential writer starvation.

A read-write lock object is an implementation-defined opaque object of type **pthread_rwlock_t** as defined in **<pthread.h>**. There are two different sorts of locks associated with a read-write lock: a *read lock* and a *write lock*.

The *pthread_rwlockattr_init*( ) function initializes a read-write lock attributes object with the default value for all the attributes defined in the implementation. After a read-write lock attributes object has been used to initialize one or more read-write locks, changes to the read-write lock attributes object, including destruction, do not affect previously initialized read-write locks.

Implementations must provide at least the read-write lock attribute *process-shared*. This attribute can have the following values:

PTHREAD_PROCESS_SHARED
Any thread of any process that has access to the memory where the read-write lock resides can manipulate the read-write lock.

PTHREAD_PROCESS_PRIVATE
Only threads created within the same process as the thread that initialized the read-write lock can manipulate the read-write lock. This is the default value.

The *pthread_rwlockattr_setpshared*( ) function is used to set the *process-shared* attribute of an initialized read-write lock attributes object while the function *pthread_rwlockattr_getpshared*( ) obtains the current value of the *process-shared* attribute.

A read-write lock attributes object is destroyed using the *pthread_rwlockattr_destroy*( ) function. The effect of subsequent use of the read-write lock attributes object is undefined.

A thread creates a read-write lock using the *pthread_rwlock_init*( ) function. The attributes of the read-write lock can be specified by the application developer; otherwise, the default implementation-defined read-write lock attributes are used if the pointer to the read-write lock attributes object is NULL. In cases where the default attributes are appropriate, the PTHREAD_RWLOCK_INITIALIZER macro can be used to initialize statically allocated read-write locks.

A thread which wants to apply a read lock to the read-write lock can use either *pthread_rwlock_rdlock*( ) or *pthread_rwlock_tryrdlock*( ). If *pthread_rwlock_rdlock*( ) is used, the thread acquires a read lock if a writer does not hold the write lock and there are no writers blocked on the write lock. If a read lock is not acquired, the calling thread blocks until it can acquire a lock. However, if *pthread_rwlock_tryrdlock*( ) is used, the function returns immediately with the error [EBUSY] if any thread holds a write lock or there are blocked writers waiting for the write lock.

A thread which wants to apply a write lock to the read-write lock can use either of two functions: *pthread_rwlock_wrlock*( ) or *pthread_rwlock_trywrlock*( ). If *pthread_rwlock_wrlock*( ) is used, the thread acquires the write lock if no other reader or writer threads hold the read-write lock. If the write lock is not acquired, the thread blocks until it can acquire the write lock. However, if *pthread_rwlock_trywrlock*( ) is used, the function returns immediately with the error [EBUSY] if any thread is holding either a read or a write lock.

The *pthread_rwlock_unlock*( ) function is used to unlock a read-write lock object held by the calling thread. Results are undefined if the read-write lock is not held by the calling thread. If there are other read locks currently held on the read-write lock object, the read-write lock object shall remain in the read locked state but without the current thread as one of its owners. If this function releases the last read lock for this read-write lock object, the read-write lock object shall be put in the unlocked read state. If this function is called to release a write lock for this read-write lock object, the read-write lock object shall be put in the unlocked state.

- Thread Concurrency Level

  On threads implementations that multiplex user threads onto a smaller set of kernel execution entities, the system attempts to create a reasonable number of kernel execution entities for the application upon application startup.

  On some implementations, these kernel entities are retained by user threads that block in the kernel. Other implementations do not *timeslice* user threads so that multiple compute-bound user threads can share a kernel thread. On such implementations, some applications may use up all the available kernel execution entities before its user-space threads are used up. The process may be left with user threads capable of doing work for the application but with no way to schedule them.

  The *pthread_setconcurrency*( ) function enables an application to request more kernel entities; that is, specify a desired concurrency level. However, this function merely provides a hint to the implementation. The implementation is free to ignore this request or to provide some other number of kernel entities. If an implementation does not multiplex user threads onto a smaller number of kernel execution entities, the *pthread_setconcurrency*( ) function has no effect.

  The *pthread_setconcurrency*( ) function may also have an effect on implementations where the kernel mode and user mode schedulers cooperate to ensure that ready user threads are not prevented from running by other threads blocked in the kernel.

  The *pthread_getconcurrency*( ) function always returns the value set by a previous call to *pthread_setconcurrency*( ). However, if *pthread_setconcurrency*( ) was not previously called, this function shall return zero to indicate that the threads implementation is maintaining the concurrency level.

- Thread Stack Guard Size

  DCE threads introduced the concept of a *thread stack guard size*. Most thread implementations add a region of protected memory to a thread's stack, commonly known as a *guard region*, as a safety measure to prevent stack pointer overflow in one thread from corrupting the contents of another thread's stack. The default size of the guard regions attribute is {PAGESIZE} bytes and is implementation-defined.

  Some application developers may wish to change the stack guard size. When an application creates a large number of threads, the extra page allocated for each stack may strain system resources. In addition to the extra page of memory, the kernel's memory manager has to keep track of the different protections on adjoining pages. When this is a problem, the application developer may request a guard size of 0 bytes to conserve system resources by eliminating stack overflow protection.

  Conversely an application that allocates large data structures such as arrays on the stack may wish to increase the default guard size in order to detect stack overflow. If a thread allocates two pages for a data array, a single guard page provides little protection against thread stack overflows since the thread can corrupt adjoining memory beyond the guard page.

The System Interfaces volume of IEEE Std 1003.1-200x defines a new attribute of a thread attributes object; that is, the *guardsize* attribute which allows applications to specify the size of the guard region of a thread's stack.

Two functions are provided for manipulating a thread's stack guard size. The *pthread_attr_setguardsize*( ) function sets the thread *guardsize* attribute, and the *pthread_attr_getguardsize*( ) function retrieves the current value.

An implementation may round up the requested guard size to a multiple of the configurable system variable {PAGESIZE}. In this case, *pthread_attr_getguardsize*( ) returns the guard size specified by the previous *pthread_attr_setguardsize*( ) function call and not the rounded up value.

If an application is managing its own thread stacks using the *stackaddr* attribute, the *guardsize* attribute is ignored and no stack overflow protection is provided. In this case, it is the responsibility of the application to manage stack overflow along with stack allocation.

- Parallel I/O

  Suppose two or more threads independently issue read requests on the same file. To read specific data from a file, a thread must first call *lseek*( ) to seek to the proper offset in the file, and then call *read*( ) to retrieve the required data. If more than one thread does this at the same time, the first thread may complete its seek call, but before it gets a chance to issue its read call a second thread may complete its seek call, resulting in the first thread accessing incorrect data when it issues its read call. One workaround is to lock the file descriptor while seeking and reading or writing, but this reduces parallelism and adds overhead.

  Instead, the System Interfaces volume of IEEE Std 1003.1-200x provides two functions to make seek/read and seek/write operations atomic. The file descriptor's current offset is unchanged, thus allowing multiple read and write operations to proceed in parallel. This improves the I/O performance of threaded applications. The *pread*( ) function is used to do an atomic read of data from a file into a buffer. Conversely, the *pwrite*( ) function does an atomic write of data from a buffer to a file.

B.2.9.1 Thread-Safety

All functions required by IEEE Std 1003.1-200x need to be thread-safe. Implementations have to provide internal synchronization when necessary in order to achieve this goal. In certain cases—for example, most floating-point implementations—context switch code may have to manage the writable shared state.

While a read from a pipe of {PIPE_MAX}*2 bytes may not generate a single atomic and thread-safe stream of bytes, it should generate ''several'' (individually atomic) thread-safe streams of bytes. Similarly, while reading from a terminal device may not generate a single atomic and thread-safe stream of bytes, it should generate some finite number of (individually atomic) and thread-safe streams of bytes. That is, concurrent calls to read for a pipe, FIFO, or terminal device are not allowed to result in corrupting the stream of bytes or other internal data. However, *read*( ), in these cases, is not required to return a single contiguous and atomic stream of bytes.

It is not required that all functions provided by IEEE Std 1003.1-200x be either async-cancel-safe or async-signal-safe.

As it turns out, some functions are inherently not thread-safe; that is, their interface specifications preclude reentrancy. For example, some functions (such as *asctime*( )) return a pointer to a result stored in memory space allocated by the function on a per-process basis. Such a function is not thread-safe, because its result can be overwritten by successive invocations.

Other functions, while not inherently non-thread-safe, may be implemented in ways that lead to them not being thread-safe. For example, some functions (such as *rand*( )) store state information (such as a seed value, which survives multiple function invocations) in memory space allocated by the function on a per-process basis. The implementation of such a function is not thread-safe if the implementation fails to synchronize invocations of the function and thus fails to protect the state information. The problem is that when the state information is not protected, concurrent invocations can interfere with one another (for example, see the same seed value).

Thread-Safety and Locking of Existing Functions

Originally, POSIX.1 was not designed to work in a multi-threaded environment, and some implementations of some existing functions will not work properly when executed concurrently. To provide routines that will work correctly in an environment with threads (''thread-safe''), two problems need to be solved:

1. Routines that maintain or return pointers to static areas internal to the routine (which may now be shared) need to be modified. The routines *ttyname*( ) and *localtime*( ) are examples.
2. Routines that access data space shared by more than one thread need to be modified. The *malloc*( ) function and the *stdio* family routines are examples.

There are a variety of constraints on these changes. The first is compatibility with the existing versions of these functions—non-thread-safe functions will continue to be in use for some time, as the original interfaces are used by existing code. Another is that the new thread-safe versions of these functions represent as small a change as possible over the familiar interfaces provided by the existing non-thread-safe versions. The new interfaces should be independent of any particular threads implementation. In particular, they should be thread-safe without depending on explicit thread-specific memory. Finally, there should be minimal performance penalty due to the changes made to the functions.

It is intended that the list of functions from POSIX.1 that cannot be made thread-safe and for which corrected versions are provided be complete.

Thread-Safety and Locking Solutions

Many of the POSIX.1 functions were thread-safe and did not change at all. However, some functions (for example, the math functions typically found in **libm**) are not thread-safe because of writable shared global state. For instance, in IEEE Std 754-1985 floating-point implementations, the computation modes and flags are global and shared.

Some functions are not thread-safe because a particular implementation is not reentrant, typically because of a non-essential use of static storage. These require only a new implementation.

Thread-safe libraries are useful in a wide range of parallel (and asynchronous) programming environments, not just within pthreads. In order to be used outside the context of pthreads, however, such libraries still have to use some synchronization method. These could either be independent of the pthread synchronization operations, or they could be a subset of the pthread interfaces. Either method results in thread-safe library implementations that can be used without the rest of pthreads.

Some functions, such as the *stdio* family interface and dynamic memory allocation functions such as *malloc*( ), are interdependent routines that share resources (for example, buffers) across related calls. These require synchronization to work correctly, but they do not require any change to their external (user-visible) interfaces.

In some cases, such as *getc*( ) and *putc*( ), adding synchronization is likely to create an unacceptable performance impact. In this case, slower thread-safe synchronized functions are to be provided, but the original, faster (but unsafe) functions (which may be implemented as macros) are retained under new names. Some additional special-purpose synchronization facilities are necessary for these macros to be usable in multi-threaded programs. This also requires changes in **<stdio.h>**.

The other common reason that functions are unsafe is that they return a pointer to static storage, making the functions non-thread-safe. This has to be changed, and there are three natural choices:

1. Return a pointer to thread-specific storage

   This could incur a severe performance penalty on those architectures with a costly implementation of the thread-specific data interface.

   A variation on this technique is to use *malloc*( ) to allocate storage for the function output and return a pointer to this storage. This technique may also have an undesirable performance impact, however, and a simplistic implementation requires that the user program explicitly free the storage object when it is no longer needed. This technique is used by some existing POSIX.1 functions. With careful implementation for infrequently used functions, there may be little or no performance or storage penalty, and the maintenance of already-standardized interfaces is a significant benefit.

2. Return the actual value computed by the function

   This technique can only be used with functions that return pointers to structures—routines that return character strings would have to wrap their output in an enclosing structure in order to return the output on the stack. There is also a negative performance impact inherent in this solution in that the output value has to be copied twice before it can be used by the calling function: once from the called routine's local buffers to the top of the stack, then from the top of the stack to the assignment target. Finally, many older compilers cannot support this technique due to a historical tendency to use internal static buffers to deliver the results of structure-valued functions.

3. Have the caller pass the address of a buffer to contain the computed value

   The only disadvantage of this approach is that extra arguments have to be provided by the calling program. It represents the most efficient solution to the problem, however, and, unlike the *malloc*( ) technique, it is semantically clear.

There are some routines (often groups of related routines) whose interfaces are inherently non-thread-safe because they communicate across multiple function invocations by means of static memory locations. The solution is to redesign the calls so that they are thread-safe, typically by passing the needed data as extra parameters. Unfortunately, this may require major changes to the interface as well.

A floating-point implementation using IEEE Std 754-1985 is a case in point. A less problematic example is the *rand48* family of pseudo-random number generators. The functions *getgrgid*( ), *getgrnam*( ), *getpwnam*( ), and *getpwuid*( ) are another such case.

The problems with *errno* are discussed in **Alternative Solutions for Per-Thread errno** (on page 3382).

Some functions can be thread-safe or not, depending on their arguments. These include the *tmpnam*( ) and *ctermid*( ) functions. These functions have pointers to character strings as arguments. If the pointers are not NULL, the functions store their results in the character string; however, if the pointers are NULL, the functions store their results in an area that may be static and thus subject to overwriting by successive calls. These should only be called by multi-thread applications when their arguments are non-NULL.

Asynchronous Safety and Thread-Safety

A floating-point implementation has many modes that effect rounding and other aspects of computation. Functions in some math library implementations may change the computation modes for the duration of a function call. If such a function call is interrupted by a signal or cancelation, the floating-point state is not required to be protected.

There is a significant cost to make floating-point operations async-cancel-safe or async-signal-safe; accordingly, neither form of async safety is required.

Functions Returning Pointers to Static Storage

For those functions that are not thread-safe because they return values in fixed size statically allocated structures, alternate ''_r'' forms are provided that pass a pointer to an explicit result structure. Those that return pointers into library-allocated buffers have forms provided with explicit buffer and length parameters.

For functions that return pointers to library-allocated buffers, it makes sense to provide ''_r'' versions that allow the application control over allocation of the storage in which results are returned. This allows the state used by these functions to be managed on an application-specific basis, supporting per-thread, per-process, or other application-specific sharing relationships.

Early proposals had provided ''_r'' versions for functions that returned pointers to variable-size buffers without providing a means for determining the required buffer size. This would have made using such functions exceedingly clumsy, potentially requiring iteratively calling them with increasingly larger guesses for the amount of storage required. Hence, *sysconf*( ) variables have been provided for such functions that return the maximum required buffer size.

Thus, the rule that has been followed by IEEE Std 1003.1-200x when adapting single-threaded non-thread-safe functions is as follows: all functions returning pointers to library-allocated storage should have ''_r'' versions provided, allowing the application control over the storage allocation. Those with variable-sized return values accept both a buffer address and a length parameter. The *sysconf*( ) variables are provided to supply the appropriate buffer sizes when required. Implementors are encouraged to apply the same rule when adapting their own existing functions to a pthreads environment.

B.2.9.2 Thread IDs

Separate programs should communicate through well-defined interfaces and should not depend on each other's implementation. For example, if a programmer decides to rewrite the *sort* program using multiple threads, it should be easy to do this so that the interface to the *sort* program does not change. Consider that if the user causes SIGINT to be generated while the *sort* program is running, keeping the same interface means that the entire sort program is killed, not just one of its threads. As another example, consider a realtime program that manages a reactor. Such a program may wish to allow other programs to control the priority at which it watches the control rods. One technique to accomplish this is to write the ID of the thread watching the control rods into a file and allow other programs to change the priority of that thread as they see fit. A simpler technique is to have the reactor process accept IPCs (Inter-Process Communication messages) from other processes, telling it at a semantic level what priority the program should assign to watching the control rods. This allows the programmer greater flexibility in the implementation. For example, the programmer can change the implementation from having one thread per rod to having one thread watching all of the rods without changing the interface. Having threads live inside the process means that the implementation of a process is invisible to outside processes (excepting debuggers and system management tools).

Threads do not provide a protection boundary. Every thread model allows threads to share memory with other threads and encourages this sharing to be widespread. This means that one thread can wipe out memory that is needed for the correct functioning of other threads that are sharing its memory. Consequently, providing each thread with its own user and/or group IDs would not provide a protection boundary between threads sharing memory.

B.2.9.3 *Thread Mutexes*

There is no additional rationale provided for this section.

B.2.9.4 *Thread Scheduling*

- Scheduling Implementation Models

  The following scheduling implementation models are presented in terms of threads and ‘‘kernel entities’’. This is to simplify exposition of the models, and it does not imply that an implementation actually has an identifiable ‘‘kernel entity’’.

  A kernel entity is not defined beyond the fact that it has scheduling attributes that are used to resolve contention with other kernel entities for execution resources. A kernel entity may be thought of as an envelope that holds a thread or a separate kernel thread. It is not a conventional process, although it shares with the process the attribute that it has a single thread of control; it does not necessarily imply an address space, open files, and so on. It is better thought of as a primitive facility upon which conventional processes and threads may be constructed.

  — System Thread Scheduling Model

  This model consists of one thread per kernel entity. The kernel entity is solely responsible for scheduling thread execution on one or more processors. This model schedules all threads against all other threads in the system using the scheduling attributes of the thread.

  — Process Scheduling Model

  A generalized process scheduling model consists of two levels of scheduling. A threads library creates a pool of kernel entities, as required, and schedules threads to run on them using the scheduling attributes of the threads. Typically, the size of the pool is a function of the simultaneously runnable threads, not the total number of threads. The kernel then schedules the kernel entities onto processors according to their scheduling attributes, which are managed by the threads library. This set model potentially allows a wide range of mappings between threads and kernel entities.

- System and Process Scheduling Model Performance

  There are a number of important implications on the performance of applications using these scheduling models. The process scheduling model potentially provides lower overhead for making scheduling decisions, since there is no need to access kernel-level information or functions and the set of schedulable entities is smaller (only the threads within the process).

  On the other hand, since the kernel is also making scheduling decisions regarding the system resources under its control (for example, CPU(s), I/O devices, memory), decisions that do not take thread scheduling parameters into account can result in unspecified delays for realtime application threads, causing them to miss maximum response time limits. |

- Rate Monotonic Scheduling

  Rate monotonic scheduling was considered, but rejected for standardization in the context of pthreads. A sporadic server policy is included.

- Scheduling Options

  In IEEE Std 1003.1-200x, the basic thread scheduling functions are defined under the Threads option, so that they are required of all threads implementations. However, there are no specific scheduling policies required by this option to allow for conforming thread implementations that are not targeted to realtime applications.

  Specific standard scheduling policies are defined to be under the Thread Execution Scheduling option, and they are specifically designed to support realtime applications by providing predictable resource sharing sequences. The name of this option was chosen to emphasize that this functionality is defined as appropriate for realtime applications that require simple priority-based scheduling.

  It is recognized that these policies are not necessarily satisfactory for some multi-processor implementations, and work is ongoing to address a wider range of scheduling behaviors. The interfaces have been chosen to create abundant opportunity for future scheduling policies to be implemented and standardized based on this interface. In order to standardize a new scheduling policy, all that is required (from the standpoint of thread scheduling attributes) is to define a new policy name, new members of the thread attributes object, and functions to set these members when the scheduling policy is equal to the new value.

Scheduling Contention Scope

In order to accommodate the requirement for realtime response, each thread has a scheduling contention scope attribute. Threads with a system scheduling contention scope have to be scheduled with respect to all other threads in the system. These threads are usually bound to a single kernel entity that reflects their scheduling attributes and are directly scheduled by the kernel.

Threads with a process scheduling contention scope need be scheduled only with respect to the other threads in the process. These threads may be scheduled within the process onto a pool of kernel entities. The implementation is also free to bind these threads directly to kernel entities and let them be scheduled by the kernel. Process scheduling contention scope allows the implementation the most flexibility and is the default if both contention scopes are supported and none is specified.

Thus, the choice by implementors to provide one or the other (or both) of these scheduling models is driven by the need of their supported application domains for worst-case (that is, realtime) response, or average-case (non-realtime) response.

Scheduling Allocation Domain

The SCHED_FIFO and SCHED_RR scheduling policies take on different characteristics on a multi-processor. Other scheduling policies are also subject to changed behavior when executed on a multi-processor. The concept of scheduling allocation domain determines the set of processors on which the threads of an application may run. By considering the application's processor scheduling allocation domain for its threads, scheduling policies can be defined in terms of their behavior for varying processor scheduling allocation domain values. It is conceivable that not all scheduling allocation domain sizes make sense for all scheduling policies on all implementations. The concept of scheduling allocation domain, however, is a useful tool for the description of multi-processor scheduling policies.

The ‘‘process control’’ approach to scheduling obtains significant performance advantages from dynamic scheduling allocation domain sizes when it is applicable.

Non-Uniform Memory Access (NUMA) multi-processors may use a system scheduling structure that involves reassignment of threads among scheduling allocation domains. In NUMA machines, a natural model of scheduling is to match scheduling allocation domains to clusters of processors. Load balancing in such an environment requires changing the scheduling allocation domain to which a thread is assigned.

Scheduling Documentation

Implementation-provided scheduling policies need to be completely documented in order to be useful. This documentation includes a description of the attributes required for the policy, the scheduling interaction of threads running under this policy and all other supported policies, and the effects of all possible values for processor scheduling allocation domain. Note that for the implementor wishing to be minimally-compliant, it is (minimally) acceptable to define the behavior as undefined.

Scheduling Contention Scope Attribute

The scheduling contention scope defines how threads compete for resources. Within IEEE Std 1003.1-200x, scheduling contention scope is used to describe only how threads are scheduled in relation to one another in the system. That is, either they are scheduled against all other threads in the system (‘‘system scope’’) or only against those threads in the process (‘‘process scope’’). In fact, scheduling contention scope may apply to additional resources, including virtual timers and profiling, which are not currently considered by IEEE Std 1003.1-200x.

Mixed Scopes

If only one scheduling contention scope is supported, the scheduling decision is straightforward. To perform the processor scheduling decision in a mixed scope environment, it is necessary to map the scheduling attributes of the thread with process-wide contention scope to the same attribute space as the thread with system-wide contention scope.

Since a conforming implementation has to support one and may support both scopes, it is useful to discuss the effects of such choices with respect to example applications. If an implementation supports both scopes, mixing scopes provides a means of better managing system-level (that is, kernel-level) and library-level resources. In general, threads with system scope will require the resources of a separate kernel entity in order to guarantee the scheduling semantics. On the other hand, threads with process scope can share the resources of a kernel entity while maintaining the scheduling semantics.

The application is free to create threads with dedicated kernel resources, and other threads that multiplex kernel resources. Consider the example of a window server. The server allocates two threads per widget: one thread manages the widget user interface (including drawing), while the other thread takes any required application action. This allows the widget to be ‘‘active’’ while the application is computing. A screen image may be built from thousands of widgets. If each of these threads had been created with system scope, then most of the kernel-level resources might be wasted, since only a few widgets are active at any one time. In addition, mixed scope is particularly useful in a window server where one thread with high priority and system scope handles the mouse so that it tracks well. As another example, consider a database server. For each of the hundreds or thousands of clients supported by a large server, an equivalent number of threads will have to be created. If each of these threads were system, the consequences would be the same as for the window server example above. However, the server could be constructed so that actual retrieval of data is done by several dedicated threads. Dedicated threads that do work for all clients frequently justify the added expense of system scope. If it were not permissible to mix system and process threads in the same process, this type of solution would not be possible.

Dynamic Thread Scheduling Parameters Access

In many time-constrained applications, there is no need to change the scheduling attributes dynamically during thread or process execution, since the general use of these attributes is to reflect directly the time constraints of the application. Since these time constraints are generally imposed to meet higher-level system requirements, such as accuracy or availability, they frequently should remain unchanged during application execution.

However, there are important situations in which the scheduling attributes should be changed. Generally, this will occur when external environmental conditions exist in which the time constraints change. Consider, for example, a space vehicle major mode change, such as the change from ascent to descent mode, or the change from the space environment to the atmospheric environment. In such cases, the frequency with which many of the sensors or acutators need to be read or written will change, which will necessitate a priority change. In other cases, even the existence of a time constraint might be temporary, necessitating not just a priority change, but also a policy change for ongoing threads or processes. For this reason, it is critical that the interface should provide functions to change the scheduling parameters dynamically, but, as with many of the other realtime functions, it is important that applications use them properly to avoid the possibility of unnecessarily degrading performance.

In providing functions for dynamically changing the scheduling behavior of threads, there were two options: provide functions to get and set the individual scheduling parameters of threads, or provide a single interface to get and set all the scheduling parameters for a given thread simultaneously. Both approaches have merit. Access functions for individual parameters allow simpler control of thread scheduling for simple thread scheduling parameters. However, a single function for setting all the parameters for a given scheduling policy is required when first setting that scheduling policy. Since the single all-encompassing functions are required, it was decided to leave the interface as minimal as possible. Note that simpler functions (such as *pthread_setprio*( ) for threads running under the priority-based schedulers) can be easily defined in terms of the all-encompassing functions.

If the *pthread_setschedparam*( ) function executes successfully, it will have set all of the scheduling parameter values indicated in *param*; otherwise, none of the scheduling parameters will have been modified. This is necessary to ensure that the scheduling of this and all other threads continues to be consistent in the presence of an erroneous scheduling parameter.

The [EPERM] error value is included in the list of possible *pthread_setschedparam*( ) error returns as a reflection of the fact that the ability to change scheduling parameters increases risks to the implementation and application performance if the scheduling parameters are changed improperly. For this reason, and based on some existing practice, it was felt that some implementations would probably choose to define specific permissions for changing either a thread's own or another thread's scheduling parameters. IEEE Std 1003.1-200x does not include portable methods for setting or retrieving permissions, so any such use of permissions is completely unspecified .

Mutex Initialization Scheduling Attributes

In a priority-driven environment, a direct use of traditional primitives like mutexes and condition variables can lead to unbounded priority inversion, where a higher priority thread can be blocked by a lower priority thread, or set of threads, for an unbounded duration of time. As a result, it becomes impossible to guarantee thread deadlines. Priority inversion can be bounded and minimized by the use of priority inheritance protocols. This allows thread deadlines to be guaranteed even in the presence of synchronization requirements.

Two useful but simple members of the family of priority inheritance protocols are the basic priority inheritance protocol and the priority ceiling protocol emulation. Under the Basic Priority

Inheritance protocol (governed by the Thread Priority Inheritance option), a thread that is blocking higher priority threads executes at the priority of the highest priority thread that it blocks. This simple mechanism allows priority inversion to be bounded by the duration of critical sections and makes timing analysis possible.

Under the Priority Ceiling Protocol Emulation protocol (governed by the Thread Priority Protection option), each mutex has a priority ceiling, usually defined as the priority of the highest priority thread that can lock the mutex. When a thread is executing inside critical sections, its priority is unconditionally increased to the highest of the priority ceilings of all the mutexes owned by the thread. This protocol has two very desirable properties in uni-processor systems. First, a thread can be blocked by a lower priority thread for at most the duration of one single critical section. Furthermore, when the protocol is correctly used in a single processor, and if threads do not become blocked while owning mutexes, mutual deadlocks are prevented.

The priority ceiling emulation can be extended to multiple processor environments, in which case the values of the priority ceilings will be assigned depending on the kind of mutex that is being used: local to only one processor, or global, shared by several processors. Local priority ceilings will be assigned the usual way, equal to the priority of the highest priority thread that may lock that mutex. Global priority ceilings will usually be assigned a priority level higher than all the priorities assigned to any of the threads that reside in the involved processors to avoid the effect called remote blocking.

Change the Priority Ceiling of a Mutex

In order for the priority protect protocol to exhibit its desired properties of bounding priority inversion and avoidance of deadlock, it is critical that the ceiling priority of a mutex be the same as the priority of the highest thread that can ever hold it, or higher. Thus, if the priorities of the threads using such mutexes never change dynamically, there is no need ever to change the priority ceiling of a mutex.

However, if a major system mode change results in an altered response time requirement for one or more application threads, their priority has to change to reflect it. It will occasionally be the case that the priority ceilings of mutexes held also need to change. While changing priority ceilings should generally be avoided, it is important that IEEE Std 1003.1-200x provide these interfaces for those cases in which it is necessary.

B.2.9.5 *Thread Cancelation*

Many existing threads packages have facilities for canceling an operation or canceling a thread. These facilities are used for implementing user requests (such as the CANCEL button in a window-based application), for implementing OR parallelism (for example, telling the other threads to stop working once one thread has found a forced mate in a parallel chess program), or for implementing the ABORT mechanism in Ada.

POSIX programs traditionally have used the signal mechanism combined with either *longjmp*( ) or polling to cancel operations. Many POSIX programmers have trouble using these facilities to solve their problems efficiently in a single-threaded process. With the introduction of threads, these solutions become even more difficult to use.

The main issues with implementing a cancelation facility are specifying the operation to be canceled, cleanly releasing any resources allocated to that operation, controlling when the target notices that it has been canceled, and defining the interaction between asynchronous signals and cancelation.

Specifying the Operation to Cancel

Consider a thread that calls through five distinct levels of program abstraction and then, inside the lowest-level abstraction, calls a function that suspends the thread. (An abstraction boundary is a layer at which the client of the abstraction sees only the service being provided and can remain ignorant of the implementation. Abstractions are often layered, each level of abstraction being a client of the lower-level abstraction and implementing a higher-level abstraction.) Depending on the semantics of each abstraction, one could imagine wanting to cancel only the call that causes suspension, only the bottom two levels, or the operation being done by the entire thread. Canceling operations at a finer grain than the entire thread is difficult because threads are active and they may be run in parallel on a multi-processor. By the time one thread can make a request to cancel an operation, the thread performing the operation may have completed that operation and gone on to start another operation whose cancelation is not desired. Thread IDs are not reused until the thread has exited, and either it was created with the *Attr detachstate* attribute set to PTHREAD_CREATE_DETACHED or the *pthread_join*( ) or *pthread_detach*( ) function has been called for that thread. Consequently, a thread cancelation will never be misdirected when the thread terminates. For these reasons, the canceling of operations is done at the granularity of the thread. Threads are designed to be inexpensive enough so that a separate thread may be created to perform each separately cancelable operation; for example, each possibly long running user request.

For cancelation to be used in existing code, cancelation scopes and handlers will have to be established for code that needs to release resources upon cancelation, so that it follows the programming discipline described in the text.

A Special Signal Versus a Special Interface

Two different mechanisms were considered for providing the cancelation interfaces. The first was to provide an interface to direct signals at a thread and then to define a special signal that had the required semantics. The other alternative was to use a special interface that delivered the correct semantics to the target thread.

The solution using signals produced a number of problems. It required the implementation to provide cancelation in terms of signals whereas a perfectly valid (and possibly more efficient) implementation could have both layered on a low-level set of primitives. There were so many exceptions to the special signal (it cannot be used with kill, no POSIX.1 interfaces can be used with it) that it was clearly not a valid signal. Its semantics on delivery were also completely different from any existing POSIX.1 signal. As such, a special interface that did not mandate the implementation and did not confuse the semantics of signals and cancelation was felt to be the better solution.

Races Between Cancelation and Resuming Execution

Due to the nature of cancelation, there is generally no synchronization between the thread requesting the cancelation of a blocked thread and events that may cause that thread to resume execution. For this reason, and because excess serialization hurts performance, when both an event that a thread is waiting for has occurred and a cancelation request has been made and cancelation is enabled, IEEE Std 1003.1-200x explicitly allows the implementation to choose between returning from the blocking call or acting on the cancelation request.

Interaction of Cancelation with Asynchronous Signals

A typical use of cancelation is to acquire a lock on some resource and to establish a cancelation cleanup handler for releasing the resource when and if the thread is canceled.

A correct and complete implementation of cancelation in the presence of asynchronous signals requires considerable care. An implementation has to push a cancelation cleanup handler on the cancelation cleanup stack while maintaining the integrity of the stack data structure. If an asynchronously generated signal is posted to the thread during a stack operation, the signal handler cannot manipulate the cancelation cleanup stack. As a consequence, asynchronous signal handlers may not cancel threads or otherwise manipulate the cancelation state of a thread. Threads may, of course, be canceled by another thread that used a *sigwait*( ) function to wait synchronously for an asynchronous signal.

In order for cancelation to function correctly, it is required that asynchronous signal handlers not change the cancelation state. This requires that some elements of existing practice, such as using *longjmp*( ) to exit from an asynchronous signal handler implicitly, be prohibited in cases where the integrity of the cancelation state of the interrupt thread cannot be ensured.

Thread Cancelation Overview

- Cancelability States

  The three possible cancelability states (disabled, deferred, and asynchronous) are encoded into two separate bits ((disable, enable) and (deferred, asynchronous)) to allow them to be changed and restored independently. For instance, short code sequences that will not block sometimes disable cancelability on entry and restore the previous state upon exit. Likewise, long or unbounded code sequences containing no convenient explicit cancelation points will sometimes set the cancelability type to asynchronous on entry and restore the previous value upon exit.

- Cancelation Points

  Cancelation points are points inside of certain functions where a thread has to act on any pending cancelation request when cancelability is enabled, if the function would block. As with checking for signals, operations need only check for pending cancelation requests when the operation is about to block indefinitely.

  The idea was considered of allowing implementations to define whether blocking calls such as *read*( ) should be cancelation points. It was decided that it would adversely affect the design of conforming applications if blocking calls were not cancelation points because threads could be left blocked in an uncancelable state.

  There are several important blocking routines that are specifically not made cancelation points:

  — *pthread_mutex_lock*( )

    If *pthread_mutex_lock*( ) were a cancelation point, every routine that called it would also become a cancelation point (that is, any routine that touched shared state would automatically become a cancelation point). For example, *malloc*( ), *free*( ), and *rand*( ) would become cancelation points under this scheme. Having too many cancelation points makes programming very difficult, leading to either much disabling and restoring of cancelability or much difficulty in trying to arrange for reliable cleanup at every possible place.

    Since *pthread_mutex_lock*( ) is not a cancelation point, threads could result in being blocked uninterruptibly for long periods of time if mutexes were used as a general synchronization mechanism. As this is normally not acceptable, mutexes should only be used to protect resources that are held for small fixed lengths of time where not being able to be canceled will not be a problem. Resources that need to be held exclusively for long periods of time should be protected with condition variables.

— *barrier_wait*( )

Canceling a barrier wait will render a barrier unusable. Similar to a barrier timeout (which the standard developers rejected), there is no way to guarantee the consistency of a barrier's internal data structures if a barrier wait is canceled.

— *pthread_spin_lock*( )

As with mutexes, spin locks should only be used to protect resources that are held for small fixed lengths of time where not being cancelable will not be a problem.

Every library routine should specify whether or not it includes any cancelation points. Typically, only those routines that may block or compute indefinitely need to include cancelation points.

Correctly coded routines only reach cancelation points after having set up a cancelation cleanup handler to restore invariants if the thread is canceled at that point. Being cancelable only at specified cancelation points allows programmers to keep track of actions needed in a cancelation cleanup handler more easily. A thread should only be made asynchronously cancelable when it is not in the process of acquiring or releasing resources or otherwise in a state from which it would be difficult or impossible to recover.

- Thread Cancelation Cleanup Handlers

The cancelation cleanup handlers provide a portable mechanism, easy to implement, for releasing resources and restoring invariants. They are easier to use than signal handlers because they provide a stack of cancelation cleanup handlers rather than a single handler, and because they have an argument that can be used to pass context information to the handler.

The alternative to providing these simple cancelation cleanup handlers (whose only use is for cleaning up when a thread is canceled) is to define a general exception package that could be used for handling and cleaning up after hardware traps and software detected errors. This was too far removed from the charter of providing threads to handle asynchrony. However, it is an explicit goal of IEEE Std 1003.1-200x to be compatible with existing exception facilities and languages having exceptions.

The interaction of this facility and other procedure-based or language-level exception facilities is unspecified in this version of IEEE Std 1003.1-200x. However, it is intended that it be possible for an implementation to define the relationship between these cancelation cleanup handlers and Ada, C++, or other language-level exception handling facilities.

It was suggested that the cancelation cleanup handlers should also be called when the process exits or calls the *exec* function. This was rejected partly due to the performance problem caused by having to call the cancelation cleanup handlers of every thread before the operation could continue. The other reason was that the only state expected to be cleaned up by the cancelation cleanup handlers would be the intraprocess state. Any handlers that are to clean up the interprocess state would be registered with *atexit*( ). There is the orthogonal problem that the *exec* functions do not honor the *atexit*( ) handlers, but resolving this is beyond the scope of IEEE Std 1003.1-200x.

- Async-Cancel Safety

  A function is said to be *async-cancel safe* if it is written in such a way that entering the function with asynchronous cancelability enabled will not cause any invariants to be violated, even if a cancelation request is delivered at any arbitrary instruction. Functions that are async-cancel-safe are often written in such a way that they need to acquire no resources for their operation and the visible variables that they may write are strictly limited.

  Any routine that gets a resource as a side-effect cannot be made async-cancel-safe (for example, *malloc*( )). If such a routine were called with asynchronous cancelability enabled, it might acquire the resource successfully, but as it was returning to the client, it could act on a cancelation request. In such a case, the application would have no way of knowing whether the resource was acquired or not.

  Indeed, because many interesting routines cannot be made async-cancel-safe, most library routines in general are not async-cancel-safe. Every library routine should specify whether or not it is async-cancel safe so that programmers know which routines can be called from code that is asynchronously cancelable.

B.2.9.6 Thread Read-Write Locks

Background

Read-write locks are often used to allow parallel access to data on multi-processors, to avoid context switches on uni-processors when multiple threads access the same data, and to protect data structures that are frequently accessed (that is, read) but rarely updated (that is, written). The in-core representation of a file system directory is a good example of such a data structure. One would like to achieve as much concurrency as possible when searching directories, but limit concurrent access when adding or deleting files.

Although read-write locks can be implemented with mutexes and condition variables, such implementations are significantly less efficient than is possible. Therefore, this synchronization primitive is included in IEEE Std 1003.1-200x for the purpose of allowing more efficient implementations in multi-processor systems.

Queuing of Waiting Threads

The *pthread_rwlock_unlock*( ) function description states that one writer or one or more readers shall acquire the lock if it is no longer held by any thread as a result of the call. However, the function does not specify which thread(s) acquire the lock, unless the Thread Execution Scheduling option is supported.

The standard developers considered the issue of scheduling with respect to the queuing of threads blocked on a read-write lock. The question turned out to be whether IEEE Std 1003.1-200x should require priority scheduling of read-write locks for threads whose execution scheduling policy is priority-based (for example, SCHED_FIFO or SCHED_RR). There are tradeoffs between priority scheduling, the amount of concurrency achievable among readers, and the prevention of writer and/or reader starvation.

For example, suppose one or more readers hold a read-write lock and the following threads request the lock in the listed order:

```
pthread_rwlock_wrlock() - Low priority thread writer_a
pthread_rwlock_rdlock() - High priority thread reader_a
pthread_rwlock_rdlock() - High priority thread reader_b
pthread_rwlock_rdlock() - High priority thread reader_c
```

When the lock becomes available, should *writer_a* block the high priority readers? Or, suppose a read-write lock becomes available and the following are queued:

```
pthread_rwlock_rdlock() - Low priority thread reader_a
pthread_rwlock_rdlock() - Low priority thread reader_b
pthread_rwlock_rdlock() - Low priority thread reader_c
pthread_rwlock_wrlock() - Medium priority thread writer_a
pthread_rwlock_rdlock() - High priority thread reader_d
```

If priority scheduling is applied then *reader_d* would acquire the lock and *writer_a* would block the remaining readers. But should the remaining readers also acquire the lock to increase concurrency? The solution adopted takes into account that when the Thread Execution Scheduling option is supported, high priority threads may in fact starve low priority threads (the application developer is responsible in this case to design the system in such a way that this starvation is avoided). Therefore, IEEE Std 1003.1-200x specifies that high priority readers take precedence over lower priority writers. However, to prevent writer starvation from threads of the same or lower priority, writers take precedence over readers of the same or lower priority.

Priority inheritance mechanisms are non-trivial in the context of read-write locks. When a high priority writer is forced to wait for multiple readers, for example, it is not clear which subset of the readers should inherit the writer's priority. Furthermore, the internal data structures that record the inheritance must be accessible to all readers, and this implies some sort of serialization that could negate any gain in parallelism achieved through the use of multiple readers in the first place. Finally, existing practice does not support the use of priority inheritance for read-write locks. Therefore, no specification of priority inheritance or priority ceiling is attempted. If reliable priority-scheduled synchronization is absolutely required, it can always be obtained through the use of mutexes.

Comparison to fcntl() Locks

The read-write locks and the *fcntl*() locks in IEEE Std 1003.1-200x share a common goal: increasing concurrency among readers, thus increasing throughput and decreasing delay.

However, the read-write locks have two features not present in the *fcntl*() locks. First, under priority scheduling, read-write locks are granted in priority order. Second, also under priority scheduling, writer starvation is prevented by giving writers preference over readers of equal or lower priority.

Also, read-write locks can be used in systems lacking a file system, such as those conforming to the minimal realtime system profile of IEEE Std 1003.13-1998.

History of Resolution Issues

Based upon some balloting objections, the draft specified the behavior of threads waiting on a read-write lock during the execution of a signal handler, as if the thread had not called the lock operation. However, this specified behavior would require implementations to establish internal signal handlers even though this situation would be rare, or never happen for many programs. This would introduce an unacceptable performance hit in comparison to the little additional functionality gained. Therefore, the behavior of read-write locks and signals was reverted back to its previous mutex-like specification.

B.2.9.7 *Thread Interactions with Regular File Operations*

There is no additional rationale provided for this section.

B.2.10 Sockets

The base document for the sockets interfaces in IEEE Std 1003.1-200x is the XNS, Issue 5.2 specification. This was primarily chosen as it aligns with IPv6. Additional material has been added from IEEE Std 1003.1g-2000, notably socket concepts, raw sockets, the *pselect*( ) function, and the **<sys/select.h>** header.

B.2.10.1 *Address Families*

There is no additional rationale provided for this section.

B.2.10.2 *Addressing*

There is no additional rationale provided for this section.

B.2.10.3 *Protocols*

There is no additional rationale provided for this section.

B.2.10.4 *Routing*

There is no additional rationale provided for this section.

B.2.10.5 *Interfaces*

There is no additional rationale provided for this section.

B.2.10.6 *Socket Types*

The type **socklen_t** was invented to cover the range of implementations seen in the field. The intent of **socklen_t** is to be the type for all lengths that are naturally bounded in size; that is, that they are the length of a buffer which cannot sensibly become of massive size: network addresses, host names, string representations of these, ancillary data, control messages, and socket options are examples. Truly boundless sizes are represented by **size_t** as in *read*( ), *write*( ), and so on.

All **socklen_t** types were originally (in BSD UNIX) of type **int**. During the development of IEEE Std 1003.1-200x, it was decided to change all buffer lengths to **size_t**, which appears at face value to make sense. When dual mode 32/64-bit systems came along, this choice unnecessarily complicated system interfaces because **size_t** (with **long**) was a different size under ILP32 and LP64 models. Reverting to **int** would have happened except that some implementations had already shipped 64-bit-only interfaces. The compromise was a type which could be defined to be any size by the implementation: **socklen_t**.

B.2.10.7 *Socket I/O Mode*

There is no additional rationale provided for this section.

B.2.10.8 *Socket Owner*

There is no additional rationale provided for this section.

B.2.10.9 *Socket Queue Limits*

There is no additional rationale provided for this section.

B.2.10.10 *Pending Error*

There is no additional rationale provided for this section.

B.2.10.11 *Socket Receive Queue*

There is no additional rationale provided for this section.

B.2.10.12 *Socket Out-of-Band Data State*

There is no additional rationale provided for this section.

B.2.10.13 *Connection Indication Queue*

There is no additional rationale provided for this section.

B.2.10.14 *Signals*

There is no additional rationale provided for this section.

B.2.10.15 *Asynchronous Errors*

There is no additional rationale provided for this section.

B.2.10.16 *Use of Options*

There is no additional rationale provided for this section.

B.2.10.17 *Use of Sockets for Local UNIX Connections*

There is no additional rationale provided for this section.

B.2.10.18 *Use of Sockets over Internet Protocols*

A raw socket allows privileged users direct access to a protocol; for example, raw access to the IP and ICMP protocols is possible through raw sockets. Raw sockets are intended for knowledgeable applications that wish to take advantage of some protocol feature not directly accessible through the other sockets interfaces.

B.2.10.19 *Use of Sockets over Internet Protocols Based on IPv4*

There is no additional rationale provided for this section.

B.2.10.20 *Use of Sockets over Internet Protocols Based on IPv6*

There is no additional rationale provided for this section.

B.2.11 Tracing

The organization of the tracing rationale differs from the traditional rationale in that this tracing rationale text is written against the trace interface as a whole, rather than against the individual components of the trace interface or the normative section in which those components are defined. Therefore the sections below do not parallel the sections of normative text in IEEE Std 1003.1-200x.

B.2.11.1 Objectives

The intended uses of tracing are application-system debugging during system development, as a ‘‘flight recorder’’ for maintenance of fielded systems, and as a performance measurement tool. In all of these intended uses, the vendor-supplied computer system and its software are, for this discussion, assumed error-free; the intent being to debug the user-written and/or third-party application code, and their interactions. Clearly, problems with the vendor-supplied system and its software will be uncovered from time to time, but this is a byproduct of the primary activity, debugging user code.

Another need for defining a trace interface in POSIX stems from the objective to provide an efficient portable way to perform benchmarks. Existing practice shows that such interfaces are commonly used in a variety of systems but with little commonality. As part of the benchmarking needs, we must consider two aspects within the trace interface.

The first, and perhaps more important one, is the qualitative aspect.

The second is the quantitative aspect.

- Qualitative Aspect

  To better understand this aspect, let us consider an example. Suppose that you want to organize a number of actions to be performed during the day. Some of these actions are known at the beginning of the day. Some others, which may be more or less important, will be triggered by reading your mail. During the day you will make some phone calls and synchronously receive some more information. Finally you will receive asynchronous phone calls that also will trigger actions. If you, or somebody else, examines your day at work, you, or he, can discover that you have not efficiently organized your work. For instance, relative to the phone calls you made, would it be preferable to make some of these early in the morning? Or to delay some others until the end of the day? Relative to the phone calls you have received, you might find that somebody you called in the morning has called you 10 times while you were performing some important work. To examine, afterwards, your day at work, you record in sequence all the trace events relative to your work. This should give you a chance of organizing your next day at work.

  This is the qualitative aspect of the trace interface. The user of a system needs to keep a trace of particular points the application passes through, so that he can eventually make some changes in the application and/or system configuration, to give the application a chance of running more efficiently.

- Quantitative Aspect

  This aspect concerns primarily realtime applications, where missed deadlines can be undesirable. Although there are, in IEEE Std 1003.1-200x, some interfaces useful for such applications (timeouts, execution time monitoring, and so on), there are no APIs to aid in the tuning of a realtime application’s behavior (**timespec** in timeouts, length of message queues, duration of driver interrupt service routine, and so on). The tuning of an application needs a means of recording timestamped important trace events during execution in order to analyze offline, and eventually, to tune some realtime features (redesign the system with less functionalities, readjust timeouts, redesign driver interrupts, and so on).

Detailed Objectives

Objectives were defined to build the trace interface and are kept for historical interest. Although some objectives are not fully respected in this trace interface, the concept of the POSIX trace interface assumes the following points:

1. It shall be possible to trace both system and user trace events concurrently.
2. It must be possible to trace per-process trace events and also to trace system trace events which are unrelated to any particular process. A per-process trace event is either user-initiated or system-initiated.
3. It must be possible to control tracing on a per process basis from either inside or outside the process.
4. It must be possible to control tracing on a per-thread basis from inside the enclosing process.
5. Trace points shall be controllable by trace event type ID from inside and outside of the process. Multiple trace points can have the same trace event type ID, and will be controlled jointly.
6. Recording of trace events is dependent on both trace event type ID and the process/thread. Both must be enabled in order to record trace events. System trace events may or may not be handled differently.
7. The API shall not mandate the ability to control tracing for more than one process at the same time.
8. There is no objective for trace control on anything bigger than a process; for example, group or session.
9. Trace propagation and control:
   a. Trace propagation across fork is optional; the default is to not trace a child process.
   b. Trace control shall span *thread_create* operations; that is, if a process is being traced, any thread will be traced as well if this thread allows tracing. The default is to allow tracing.
10. Trace control shall not span *exec* or *spawn* operations.
11. A triggering API is not required. The triggering API is the ability to command or stop tracing based on the occurrence of specific trace event other than a POSIX_TRACE_START trace event or a POSIX_TRACE_STOP trace event.
12. Trace log entries shall have timestamps of implementation-defined resolution. Implementations are exhorted to support at least microsecond resolution. When a trace log entry is retrieved, it shall have timestamp, PC address, PID, and TID of the entity that generated the trace event.
13. Independently developed code should be able to use trace facilities without coordination and without conflict.
14. Even if the trace points in the trace calls are not unique, the trace log entries (after any processing) shall be uniquely identified as to trace point.
15. There shall be a standard API to read the trace stream.

16. The format of the trace stream and the trace log is opaque and unspecified.
17. It shall be possible to read a completed trace, if recorded on some suitable non-volatile storage, even subsequent to a power cycle or subsequent cold boot of the system.
18. Support of analysis of a trace log while it is being formed is implementation-defined.
19. The API shall allow the application to write trace stream identification information into the trace stream and to be able to retrieve it, without it being overwritten by trace entries, even if the trace stream is full.
20. It must be possible to specify the destination of trace data produced by trace events.
21. It must be possible to have different trace streams, and for the tracing enabled by one trace stream to be completely independent of the tracing of another trace stream.
22. It must be possible to trace events from threads in different CPUs.
23. The API shall support one or more trace streams per-system, and one or more trace streams per-process, up to an implementation-defined set of per-system and per-process maximums.
24. It shall be possible to determine the order in which the trace events happened, without necessarily depending on the clock, up to an implementation-defined time resolution.
25. For performance reasons, the trace event point call(s) shall be implementable as a macro (see the ISO POSIX-1: 1996 standard, 1.3.4, Statement 2).
26. IEEE Std 1003.1-200x must not define the trace points which a conforming system must implement, except for trace points used in the control of tracing.
27. The APIs shall be thread-safe, and trace points should be lock-free (that is shall not require a lock to gain exclusive access to some resource).
28. The user-provided information associated with a trace event is variable-sized, up to some maximum size.
29. Bounds on record and trace stream sizes:
    a. The API must permit the application to declare the upper bounds on the length of an application data record. The system shall return the limit it used. The limit used may be smaller than requested.
    b. The API must permit the application to declare the upper bounds on the size of trace streams. The system shall return the limit it used. The limit used may be different, either larger or smaller, than requested.
30. The API must be able to pass any fundamental data type, and a structured data type composed only of fundamental types. The API must be able to pass data by reference, given only as an address and a length. Fundamental types are the POSIX.1 types (see the **<sys/types.h>** header) plus those defined in the ISO C standard.
31. The API shall apply the POSIX notions of ownership and permission to recorded trace data, corresponding to the sources of that data.

Comments on Objectives

Note: In the following comments, numbers in square brackets refer to the above objectives.

It is necessary to be able to obtain a trace stream for a complete activity. This means we need to be able to trace both application and system trace events. A per-process trace event is either user-initiated, like the *write*( ) POSIX call, or system-initiated, like a timer expiration. We also need to be able to trace an entire process's activity even when it has threads in multiple CPUs. To avoid excess trace activity, it is necessary to be able to control tracing on a trace event type basis.
[Objectives 1,2,5,22]

We need to be able to control tracing on a per-process basis, both from inside and outside the process; that is, a process can start a trace activity on itself or any other process. We also see the need to allow the definition of a maximum number trace streams per system.
[Objectives 3,23]

From within a process, it is necessary to be able to control tracing on a per-thread basis. This provides an additional filtering capability to keep the amount of traced data to a minimum. It also allows for less ambiguity as to the origin of trace events. It is recognized that thread-level control is only valid from within the process itself. It is also desirable to know the maximum number of trace streams per process that can be started. We do not want the API to require thread synchronization or to mandate priority inversions that would cause the thread to block. However, the API must be thread-safe.
[Objectives 4,23,24,27]

We see no objective to control tracing on anything larger than a process; for example, a group or session. Also, the ability to start or stop a trace activity on multiple processes atomically may be very difficult or cumbersome in some implementations.
[Objectives 6,8]

It is also necessary to be able to control tracing by trace event type identifier, sometimes called a trace hook ID. However, there is no mandated set of system trace events, since such trace points are very system-dependent. The API must not require from the operating system facilities that are not standard (POSIX).
[Objectives 6,26]

Trace control must span *fork*( ) and *pthread_create*( ). If not, there will be no way to ensure that a program's activity is entirely traced. The newly forked child would not be able to turn on its tracing until after it obtained control after the fork, and trace control externally would be even more problematic.
[Objective 9]

Since *exec*( ) and *spawn*( ) represent a complete change in the execution of a task (a new program), trace control need not persist over an *exec*( ) or *spawn*( ).
[Objective 10]

Where trace activities are started on multiple processes, these trace activities should not interfere with each other.
[Objective 21]

There is no need for a triggering objective, primarily for performance reasons; see also Section B.2.11.8 (on page 3491), rationale on triggering.
[Objective 11]

It must be possible to determine the origin of each traced event. We need the process and thread identifiers for each trace event. We also saw the need for a user-specifiable origin, but felt this would create too much overhead.
[Objectives 12,14]

We must allow for trace points to come embedded in software components from several different sources and vendors without requiring coordination.
[Objective 13]

We need to be able to uniquely identify trace points that may have the same trace stream identifier. We only need to be able to do this when a trace report is produced.
[Objectives 12,14]

Tracing is a very performance-sensitive activity, and will therefore likely be implemented at a low level within the system. Hence the interface shall not mandate any particular buffering or storage method. Therefore, we will need a standard API to read a trace stream. Also the interface shall not mandate the format of the trace data, and the interface shall not assume a trace storage method. Due to the possibility of a monolithic kernel and the possible presence of multiple processes capable of running trace activities, the two kinds of trace events may be stored in two separate streams for performance reasons. A mandatory dump mechanism, common in some existing practice, has been avoided to allow the implementation of this set of functions on small realtime profiles for which the concept of a file system is not defined. The trace API calls should be implemented as macros.
[Objectives 15,16,25,30]

Since a trace facility is a valuable service tool, the output (or log) of a completed trace stream that is written to permanent storage must be readable on other systems of the type that produced the trace log. Note that there is no objective to be able to interpret a trace log that was not successfully completed.
[Objectives 17,18,19]

For trace streams written to permanent storage, a way to specify the destination of the trace stream is needed.
[Objective 20]

We need to be able to depend on the ordering of trace events up to some implementation-defined time interval. For example, we need to know the time period which, if trace events are closer together, their ordering is unspecified. Events that occur within an interval smaller than this resolution may or may not be read back in the correct order.
[Objective 24]

The application should be able to know how much data can be traced. When trace event types can be filtered, the application should be able to specify the approximate maximum amount of data that will be traced in a trace event so resources can be more efficiently allocated.
[Objectives 28,29]

Users should not be able to trace data to which they would not normally have access to. System trace events corresponding to a process / thread should be associated with the ownership of that process / thread.
[Objective 31]

B.2.11.2 Trace Model

Introduction

The model is based on two base entities: the ‘‘Trace Stream’’ and the ‘‘Trace Log’’ , and a recorded unit called the ‘‘Trace Event’’. The possibility of using Trace Streams and Trace Logs separately gives us two use dimensions and solves both the performance issue and the full-information system issue. In the case of a trace stream without log, specific information, although reduced in quantity, is required to be registered, in a possibly small realtime system, with as little overhead as possible. The Trace Log option has been added for small realtime systems. In the case of a trace stream with log, considerable complex application-specific information needs to be collected.

Trace Model Description

The trace model can be examined for three different subfunctions: Application Instrumentation, Trace Operation Control, and Trace Analysis.

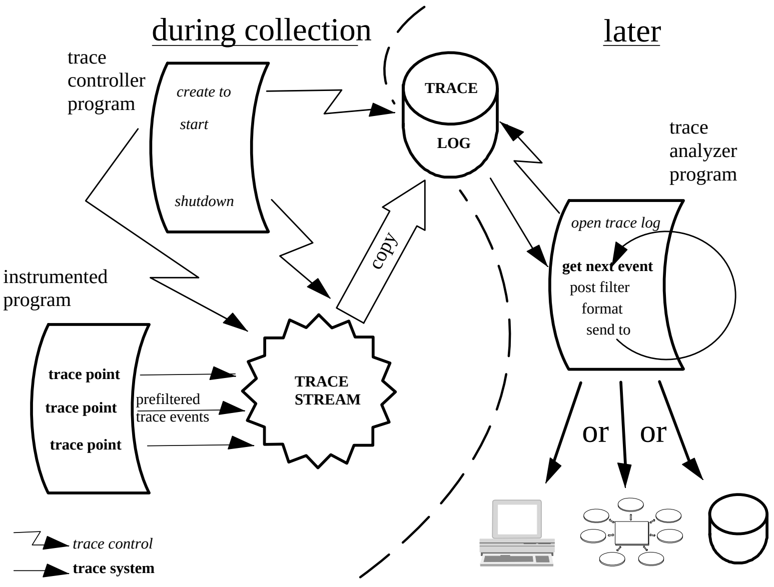


Figure B-2 Trace System Overview: for Offline Analysis

Each of these subfunctions requires specific characteristics of the trace mechanism API.

- Application Instrumentation

  When instrumenting an application, the programmer has no concern about the future utilization of the trace events in trace stream or trace log, the full policy of trace stream, or the eventual pre-filtering of trace events. But he is concerned about the correct determination of specific trace event type identifier, regardless of how many independent libraries are used in the same user application; see Figure B-2 and Figure B-3 (on page 3475).

This trace API shall provide the necessary operations to accomplish this subfunction. This is done by providing functions to associate a programmer-defined name with an implementation-defined trace event type identifier; see the *posix_trace_eventid_open*( ) function), and to send this trace event into a potential trace stream (see the *posix_trace_event*( ) function).

- Trace Operation Control

  When controlling the recording of trace events in a trace stream, the programmer is concerned with the correct initialization of the trace mechanism (that is, the sizing of the trace stream), the correct retention of trace events in a permanent storage, the correct dynamic recording of trace events, and so on.

  This trace API shall provide the necessary material to permit this efficiently. This is done by providing functions to initialize a new trace stream, and optionally a trace log:

  — Trace Stream Attributes Object Initialization (see *posix_trace_attr_init*( ))

  — Functions to Retrieve or Set Information About a Trace Stream (see *posix_trace_attr_getgenversion*( ))

  — Functions to Retrieve or Set the Behavior of a Trace Stream (see *posix_trace_attr_getinherited*( ))

  — Functions to Retrieve or Set Trace Stream Size Attributes (see *posix_trace_attr_getmaxusereventsize*( ))

  — Trace Stream Initialization, Flush, and Shutdown from a Process (see *posix_trace_create*( ))

  — Clear Trace Stream and Trace Log (see *posix_trace_clear*( ))

  To select the trace event types that are to be traced:

  — Manipulate Trace Event Type Identifier (see *posix_trace_trid_eventid_open*( ))

  — Iterate over a Mapping of Trace Event Type (see *posix_trace_eventtypelist_getnext_id*( ))

  — Manipulate Trace Event Type Sets (see *posix_trace_eventset_empty*( ))

  — Set Filter of an Initialized Trace Stream (see *posix_trace_set_filter*( ))

  To control the execution of an active trace stream:

  — Trace Start and Stop (see *posix_trace_start*( ))

  — Functions to Retrieve the Trace Attributes or Trace Statuses (see *posix_trace_get_attr*( ))

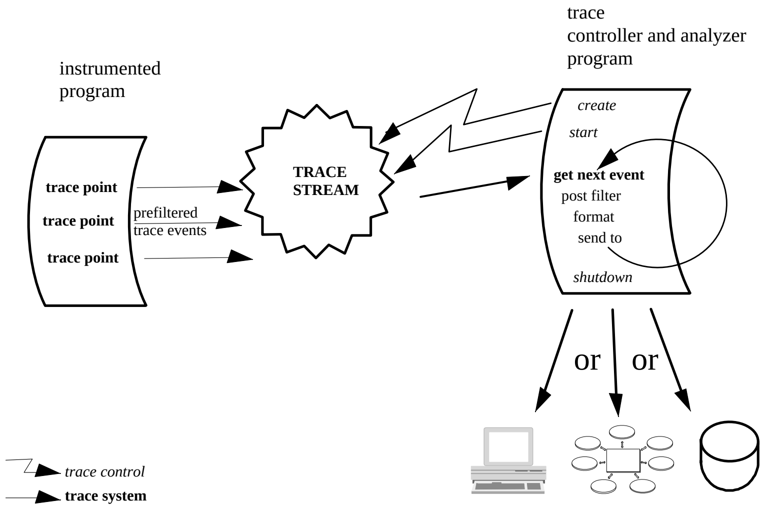


Figure B-3 Trace System Overview: for Online Analysis

- Trace Analysis

  Once correctly recorded, on permanent storage or not, an ultimate activity consists of the analysis of the recorded information. If the recorded data is on permanent storage, a specific open operation is required to associate a trace stream to a trace log.

  The first intent of the group was to request the presence of a system identification structure in the trace stream attribute. This was, for the application, to allow some portable way to process the recorded information. However, there is no requirement that the **utsname** structure, on which this system identification was based, be portable from one machine to another, so the contents of the attribute cannot be interpreted correctly by an application conforming to IEEE Std 1003.1-200x.

  Draft 6 incorporates this modification and requests that some unspecified information be recorded in the trace log in order to fail opening it if the analysis process and the controller process were running in different types of machine, but does not request that this information be accessible to the application. This modification has implied a modification in the *posix_trace_open*( ) function error code returns.

  This trace API shall provide functions to:

  — Extract trace stream identification attributes (see *posix_trace_attr_getgenversion*( ))

  — Extract trace stream behavior attributes (see *posix_trace_attr_getinherited*( ))

  — Extract trace event, stream, and log size attributes (see *posix_trace_attr_getmaxusereventsize*( ))

  — Look up trace event type names (see *posix_trace_eventid_get_name*( ))

- Iterate over trace event type identifiers (see *posix_trace_eventtypelist_getnext_id*( ))
- Open, rewind, and close a trace log (see *posix_trace_open*( ))
- Read trace stream attributes and status (see *posix_trace_get_attr*( ))
- Read trace events (see *posix_trace_getnext_event*( ))

Due to the following two reasons:

1. The requirement that the trace system must not add unacceptable overhead to the traced process and so that the trace event point execution must be fast
2. The traced application does not care about tracing errors

the trace system cannot return any internal error to the application. Internal error conditions can range from unrecoverable errors that will force the active trace stream to abort, to small errors that can affect the quality of tracing without aborting the trace stream. The group decided to define a system trace event to report to the analysis process such internal errors. It is not the intention of IEEE Std 1003.1-200x to require an implementation to report an internal error that corrupts or terminates tracing operation. The implementor is free to decide which internal documented errors, if any, the trace system is able to report.

States of a Trace Stream

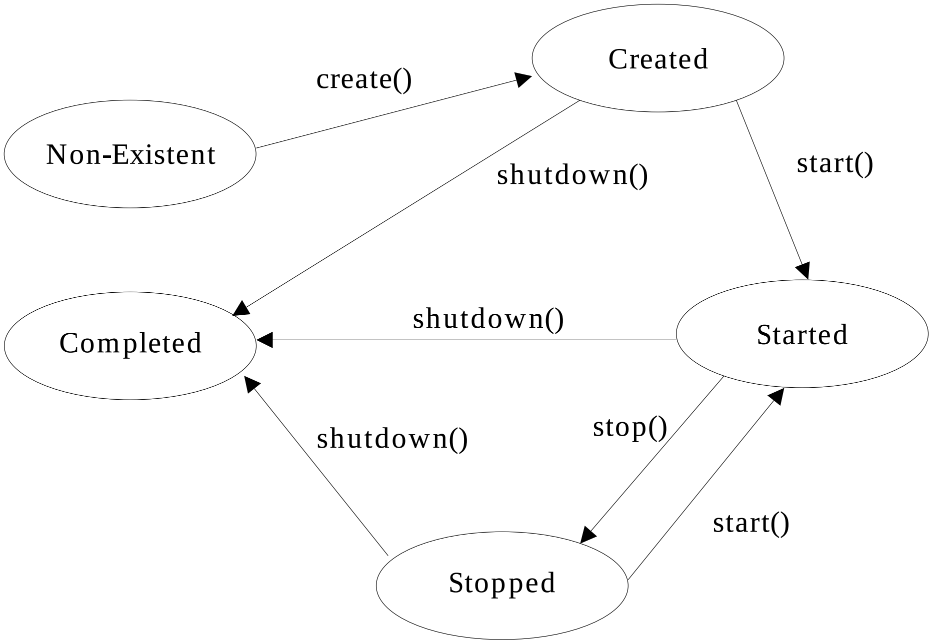


Figure B-4 Trace System Overview: States of a Trace Stream

Figure B-4 shows the different states an active trace stream passes through. After the *posix_trace_create*( ) function call, a trace stream becomes CREATED and a trace stream is associated for the future collection of trace events. The status of the trace stream is POSIX_TRACE_SUSPENDED. The state becomes STARTED after a call to the *posix_trace_start*( ) function, and the status becomes POSIX_TRACE_RUNNING. In this state, all trace events that are not filtered out shall be stored into the trace stream. After a call to *posix_trace_stop*( ), the trace stream becomes STOPPED (and the status POSIX_TRACE_SUSPENDED). In this state, no new trace events will be recorded in the trace stream, but previously recorded trace events may continue to be read.

After a call to *posix_trace_shutdown*( ), the trace stream is in the state COMPLETED. The trace stream no longer exists but, if the Trace Log option is supported, all the information contained in it has been logged. If a log object has not been associated with the trace stream at the creation, it is the responsibility of the trace controller process to not shut the trace stream down while trace events remain to be read in the stream.

Tracing All Processes

Some implementations have a tracing subsystem with the ability to trace all processes. This is useful to debug some types of device drivers such as those for ATM or X25 adapters. These types of adapters are used by several independent processes, that are not issued from the same process.

The POSIX trace interface does not define any constant or option to create a trace stream tracing all processes. But the POSIX trace interface does not prevent this type of implementation and the implementor is free to add this capability. Nevertheless, the POSIX trace interface allows to trace all the system trace events and all the processes issued from the same process.

If such a tracing system capability has to be implemented, when a trace stream is created, it is recommended that a constant named POSIX_TRACE_ALLPROC be used instead of the process identifier in the argument of the function *posix_trace_create*( ) or *posix_trace_create_withlog*( ). A possible value for POSIX_TRACE_ALLPROC may be −1 instead of a real process identifier.

The implementor has to be aware that there is some impact on the tracing behavior as defined in the POSIX trace interface. For example:

- If the default value for the inheritance attribute is to set to POSIX_TRACE_CLOSE_FOR_CHILD, the implementation has to stop tracing for the child process.
- The trace controller which is creating this type of trace stream must have the appropriate privilege to trace all the processes.

Trace Storage

The model is based on two types of trace events: system trace events and user-defined trace events. The internal representation of trace events is implementation-defined, and so the implementor is free to choose the more suitable, practical, and efficient way to design the internal management of trace events. For the timestamping operation, the model does not impose the CLOCK_REALTIME or any other clock. The buffering allocation and operation follow the same principle. The implementor is free to use one or more buffers to record trace events; the interface assumes only a logical trace stream of sequentially recorded trace events. Regarding flushing of trace events, the interface allows the definition of a trace log object which typically can be a file. But the group was also aware of defining functions to permit the use of this interface in small realtime systems, which may not have general file system capabilities. For instance, the three functions *posix_trace_getnext_event*( ) (blocking), *posix_trace_timedgetnext_event*( ) (blocking with timeout), and *posix_trace_trygetnext_event*( ) (non-blocking) are proposed to read the recorded trace events.

The policy to be used when the trace stream becomes full also relies on common practice:

- For an active trace stream, the POSIX_TRACE_LOOP trace stream policy permits automatic overrun (overwrite of oldest trace events) while waiting for some user-defined condition to cause tracing to stop. By contrast, the POSIX_TRACE_UNTIL_FULL trace stream policy requires the system to stop tracing when the trace stream is full. However, if the trace stream that is full is at least partially emptied by a call to the *posix_trace_flush*( ) function or by calls to *posix_trace_getnext_event*( ) function, the trace system will automatically resume tracing.

If the Trace Log option is supported the operation of the POSIX_TRACE_FLUSH policy is an extension of the POSIX_TRACE_UNTIL_FULL policy. The automatic free operation (by flushing to the associated trace log) is added.

- If a log is associated with the trace stream and this log is a regular file, these policies also apply for the log. One more policy, POSIX_TRACE_APPEND, is defined to allow indefinite extension of the log. Since the log destination can be any device or pseudo-device, the implementation may not be able to manipulate the destination as required by IEEE Std 1003.1-200x. For this reason, the behavior of the log full policy may be unspecified depending of the trace log type.

  The current trace interface does not define a service to preallocate space for a trace log file, because this space can be preallocated by means of a call to the *posix_fallocate*( ) function. This function could be called after the file has been opened, but before the trace stream is created. The *posix_fallocate*( ) function ensures that any required storage for regular file data is allocated on the file system storage media. If *posix_fallocate*( ) returns successfully, subsequent writes to the specified file data shall not fail due to the lack of free space on the file system storage media. Besides trace events, a trace stream also includes trace attributes and the mapping from trace event names to trace event type identifiers. The implementor is free to choose how to store the trace attributes and the trace event type map, but must ensure that this information is not lost when a trace stream overrun occurs.

B.2.11.3 Trace Programming Examples

Several programming examples are presented to show the code of the different possible subfunctions using a trace subsystem. All these programs need to include the **<trace.h>** header. In the examples shown, error checking is omitted for more simplicity.

Trace Operation Control

These examples show the creation of a trace stream for another process; one which is already trace instrumented. All the default trace stream attributes are used to simplify programming in the first example. The second example shows more possibilities.

First Example

```
/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    pid_t pid = traced_process_pid;
    int fd;
    trace_id_t trid;

    - - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Open a trace log */
    fd=open("/tmp/mytracelog",...);
    /*
     * Create a new trace associated with a log
     * and with default attributes
     */
```

```
        posix_trace_create_withlog(pid, &attr, fd, &trid);
        /* Trace attribute structure can now be destroyed */
        posix_trace_attr_destroy(&attr);
        /* Start of trace event recording */
        posix_trace_start(trid);
        - - - - - -
        - - - - - -
        /* Duration of tracing */
        - - - - - -
        - - - - - -
        /* Stop and shutdown of trace activity */
        posix_trace_shutdown(trid);
        - - - - - -
    }
```

Second Example

Between the initialization of the trace stream attributes and the creation of the trace stream, these trace stream attributes may be modified; see **Trace Stream Attribute Manipulation** (on page 3483) for specific programming example. Between the creation and the start of the trace stream, the event filter may be set; after the trace stream is started, the event filter may be changed. The setting of an event set and the change of a filter is shown in **Create a Trace Event Type Set and Change the Trace Event Type Filter** (on page 3483).

```
    /* Caution. Error checks omitted */
    {
        trace_attr_t attr;
        pid_t pid = traced_process_pid;
        int fd;
        trace_id_t trid;
        - - - - - -
        /* Initialize trace stream attributes */
        posix_trace_attr_init(&attr);
        /* Attr default may be changed at this place; see example */
        - - - - - -
        /* Create and open a trace log with R/W user access */
        fd=open("/tmp/mytracelog",O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
        /* Create a new trace associated with a log */
        posix_trace_create_withlog(pid, &attr, fd, &trid);
        /*
         * If the Trace Filter option is supported
         * trace event type filter default may be changed at this place;
         * see example about changing the trace event type filter
         */
        posix_trace_start(trid);
        - - - - - -

        /*
         * If you have an uninteresting part of the application
         * you can stop temporarily.
         *
         * posix_trace_stop(trid);
         * - - - - - -
```

```
         * - - - - - -
         * posix_trace_start(trid);
         */
        - - - - - -
        /*
         * If the Trace Filter option is supported
         * the current trace event type filter can be changed
         * at any time (see example about how to set
         * a trace event type filter
         */
        - - - - - -

        /* Stop the recording of trace events */
        posix_trace_stop(trid);
        /* Shutdown the trace stream */
        posix_trace_shutdown(trid);
        /*
         * Destroy trace stream attributes; attr structure may have
         * been used during tracing to fetch the attributes
         */
        posix_trace_attr_destroy(&attr);
        - - - - - -
}
```

Application Instrumentation

This example shows an instrumented application. The code is included in a block of instructions, perhaps a function from a library. Possibly in an initialization part of the instrumented application, two user trace events names are mapped to two trace event type identifiers (function *posix_trace_eventid_open*( )). Then two trace points are programmed.

```
/* Caution. Error checks omitted */
{
    trace_event_id_t eventid1, eventid2;
    - - - - - -
    /* Initialization of two trace event type ids */
    posix_trace_eventid_open("my_first_event",&eventid1);
    posix_trace_eventid_open("my_second_event",&eventid2);
    - - - - - -
    - - - - - -
    - - - - - -
    /* Trace point */
    posix_trace_event(eventid1,NULL,0);
    - - - - - -
    /* Trace point */
    posix_trace_event(eventid2,NULL,0);
    - - - - - -
}
```

Trace Analyzer

This example shows the manipulation of a trace log resulting from the dumping of a completed trace stream. All the default attributes are used to simplify programming, and data associated with a trace event are not shown in the first example. The second example shows more possibilities.

First Example

```
/* Caution. Error checks omitted */
{
    int fd;
    trace_id_t trid;
    posix_trace_event_info trace_event;
    char trace_event_name[TRACE_EVENT_NAME_MAX];
    int return_value;
    size_t returndatasize;
    int lost_event_number;

    - - - - - -

    /* Open an existing trace log */
    fd=open("/tmp/tracelog", O_RDONLY);
    /* Open a trace stream on the open log */
    posix_trace_open(fd, &trid);
    /* Read a trace event */
    posix_trace_getnext_event(trid, &trace_event,
        NULL, 0, &returndatasize,&return_value);

    /* Read and print all trace event names out in a loop */
    while (return_value == NULL)
    {
        /*
         * Get the name of the trace event associated
         * with trid trace ID
         */
        posix_trace_eventid_get_name(trid, trace_event.event_id,
            trace_event_name);
        /* Print the trace event name out */
        printf("%s\n",trace_event_name);
        /* Read a trace event */
        posix_trace_getnext_event(trid, &trace_event,
            NULL, 0, &returndatasize,&return_value);
    }

    /* Close the trace stream */
    posix_trace_close(trid);
    /* Close the trace log */
    close(fd);
}
```

Second Example

The complete example includes the two other examples in **Retrieve Information from a Trace Log** (on page 3484) and in **Retrieve the List of Trace Event Types Used in a Trace Log** (on page 3485). For example, the *maxdatasize* variable is set in **Retrieve the List of Trace Event Types Used in a Trace Log** (on page 3485).

```
/* Caution. Error checks omitted */
{
    int fd;
    trace_id_t trid;
    posix_trace_event_info trace_event;
    char trace_event_name[TRACE_EVENT_NAME_MAX];
    char * data;
    size_t maxdatasize=1024, returndatasize;
    int return_value;
    - - - - - -

    /* Open an existing trace log */
    fd=open("/tmp/tracelog", O_RDONLY);
    /* Open a trace stream on the open log */
    posix_trace_open( fd, &trid);
    /*
     * Retrieve information about the trace stream which
     * was dumped in this trace log (see example)
     */
    - - - - - -

    /* Allocate a buffer for trace event data */
    data=(char *)malloc(maxdatasize);
    /*
     * Retrieve the list of trace event used in this
     * trace log (see example)
     */
    - - - - - -

    /* Read and print all trace event names and data out in a loop */
    while (1)
    {
    posix_trace_getnext_event(trid, &trace_event,
        data, maxdatasize, &returndatasize,&return_value);
        if (return_value != NULL) break;
        /*
         * Get the name of the trace event type associated
         * with trid trace ID
         */
        posix_trace_eventid_get_name(trid, trace_event.event_id,
            trace_event_name);
        {
        int i;

        /* Print the trace event name out */
        printf("%s: ", trace_event_name);
        /* Print the trace event data out */
        for (i=0; i<returndatasize, i++) printf("%02.2X",
```

```
                (unsigned char)data[i]);
            printf("\n");
            }
        }
        /* Close the trace stream */
        posix_trace_close(trid);
        /* The buffer data is deallocated */
        free(data);
        /* Now the file can be closed */
        close(fd);
    }
```

Several Programming Manipulations

The following examples show some typical sets of operations needed in some contexts.

Trace Stream Attribute Manipulation

This example shows the manipulation of a trace stream attribute object in order to change the default value provided by a previous *posix_trace_attr_init*( ) call.

```
/* Caution. Error checks omitted */
{
    trace_attr_t attr;
    size_t logsize=100000;
    - - - - - -
    /* Initialize trace stream attributes */
    posix_trace_attr_init(&attr);
    /* Set the trace name in the attributes structure */
    posix_trace_attr_setname(&attr, "my_trace");
    /* Set the trace full policy */
    posix_trace_attr_setstreamfullpolicy(&attr, POSIX_TRACE_LOOP);
    /* Set the trace log size */
    posix_trace_attr_setlogsize(&attr, logsize);
    - - - - - -
}
```

Create a Trace Event Type Set and Change the Trace Event Type Filter

This example is valid only if the Trace Event Filter option is supported. This example shows the manipulation of a trace event type set in order to change the trace event type filter for an existing active trace stream, which may be just-created, running, or suspended. Some sets of trace event types are well-known, such as the set of trace event types not associated with a process, some trace event types are just-built trace event types for this trace stream; one trace event type is the predefined trace event error type which is deleted from the trace event type set.

```
/* Caution. Error checks omitted */
{
    trace_id_t trid = existing_trace;
    trace_event_set_t set;
    trace_event_id_t trace_event1, trace_event2;
    - - - - - -
    /* Initialize to an empty set of trace event types */
```

```
    /* (not strictly required because posix_trace_event_set_fill() */
    /* will ignore the prior contents of the event set.) */
    posix_trace_eventset_emptyset(&set);
    /*
     * Fill the set with all system trace events
     * not associated with a process
     */
    posix_trace_eventset_fill(&set, POSIX_TRACE_WOPID_EVENTS);

    /*
     * Get the trace event type identifier of the known trace event name
     * my_first_event for the trid trace stream
     */
    posix_trace_trid_eventid_open(trid, "my_first_event", &trace_event1);
    /* Add the set with this trace event type identifier */
    posix_trace_eventset_add_event(trace_event1, &set);
    /*
     * Get the trace event type identifier of the known trace event name
     * my_second_event for the trid trace stream
     */

    posix_trace_trid_eventid_open(trid, "my_second_event", &trace_event2);
    /* Add the set with this trace event type identifier */
    posix_trace_eventset_add_event(trace_event2, &set);
    - - - - - -
    /* Delete the system trace event POSIX_TRACE_ERROR from the set */
    posix_trace_eventset_del_event(POSIX_TRACE_ERROR, &set);
    - - - - - -

    /* Modify the trace stream filter making it equal to the new set */
    posix_trace_set_filter(trid, &set, POSIX_TRACE_SET_EVENTSET);
    - - - - - -
    /*
     * Now trace_event1, trace_event2, and all system trace event types
     * not associated with a process, except for the POSIX_TRACE_ERROR
     * system trace event type, are filtered out of (not recorded in) the
     * existing trace stream.
     */
}
```

Retrieve Information from a Trace Log

This example shows how to extract information from a trace log, the dump of a trace stream. This code:

- Asks if the trace stream has lost trace events
- Extracts the information about the version of the trace subsystem which generated this trace log
- Retrieves the maximum size of trace event data; this may be used to dynamically allocate an array for extracting trace event data from the trace log without overflow

```
/* Caution. Error checks omitted */
{
    struct posix_trace_status_info statusinfo;
```

```
        trace_attr_t attr;
        trace_id_t trid = existing_trace;
        size_t maxdatasize;
        char genversion[TRACE_NAME_MAX];
        - - - - - -
        /* Get the trace stream status */
        posix_trace_get_status(trid, &statusinfo);
        /* Detect an overrun condition */
        if (statusinfo.posix_stream_overrun_status == POSIX_TRACE_OVERRUN)
            printf("trace events have been lost\n");

        /* Get attributes from the trid trace stream */
        posix_trace_get_attr(trid, &attr);
        /* Get the trace generation version from the attributes */
        posix_trace_attr_getgenversion(&attr, genversion);
        /* Print the trace generation version out */
        printf("Information about Trace Generator:%s\n",genversion);

        /* Get the trace event max data size from the attributes */
        posix_trace_attr_getmaxdatasize(&attr, &maxdatasize);
        /* Print the trace event max data size out */
        printf("Maximum size of associated data:%d\n",maxdatasize);
        /* Destroy the trace stream attributes */
        posix_trace_attr_destroy(&attr);
    }
```

Retrieve the List of Trace Event Types Used in a Trace Log

This example shows the retrieval of a trace stream's trace event type list. This operation may be very useful if you are interested only in tracking the type of trace events in a trace log.

```
/* Caution. Error checks omitted */
{
    trace_id_t trid = existing_trace;
    trace_event_id_t event_id;
    char event_name[TRACE_EVENT_NAME_MAX];
    int return_value;
    - - - - - -

    /*
     * In a loop print all existing trace event names out
     * for the trid trace stream
     */
    while (1)
    {
        posix_trace_eventtypelist_getnext_id(trid, &event_id
            &return_value);
        if (return_value != NULL) break;
        /*
         * Get the name of the trace event associated
         * with trid trace ID
         */
        posix_trace_eventid_get_name(trid, event_id, event_name);
        /* Print the name out */
```

```
                printf("%s\n", event_name);
            }
        }
```

B.2.11.4 Rationale on Trace for Debugging

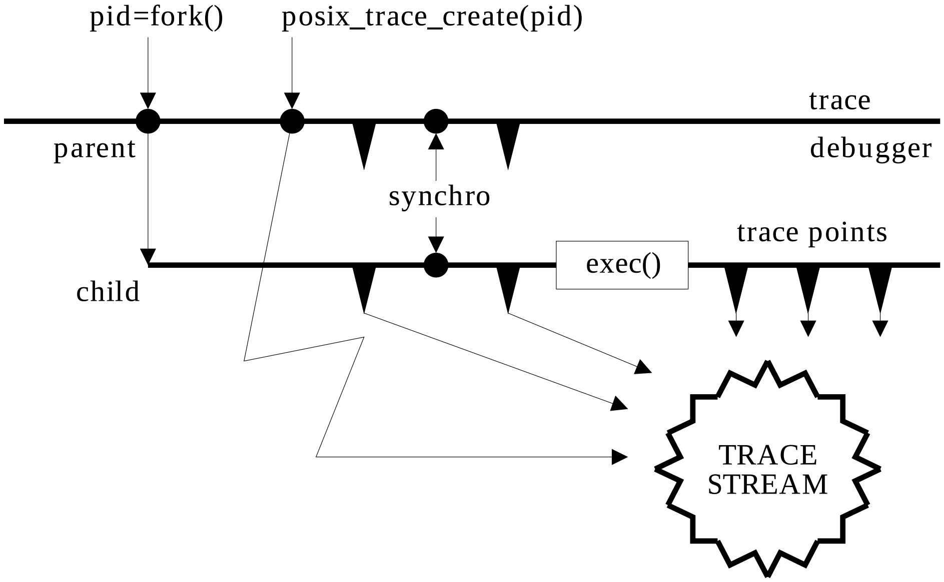


Figure B-5 Trace Another Process

Among the different possibilities offered by the trace interface defined in IEEE Std 1003.1-200x, the debugging of an application is the most interesting one. Typical operations in the controlling debugger process are to filter trace event types, to get trace events from the trace stream, to stop the trace stream when the debugged process is executing uninteresting code, to start the trace stream when some interesting point is reached, and so on. The interface defined in IEEE Std 1003.1-200x should define all the necessary base functions to allow this dynamic debug handling.

Figure B-5 shows an example in which the trace stream is created after the call to the *fork*( ) function. If the user does not want to lose trace events some synchronization mechanism (represented in the figure) may be needed before calling the *exec*( ) function, to give the parent a chance to create the trace stream before the child begins the execution of its trace points.

B.2.11.5 Rationale on Trace Event Type Name Space

At first, the working group was in favor of the representation of a trace event type by an integer (*event_name*). It seems that existing practice shows the weakness of such a representation. The collision of trace event types is the main problem that cannot be simply resolved using this sort of representation. Suppose, for example, that a third party designs an instrumented library. The user does not have the source of this library and wants to trace his application which uses in some part the third-party library. There is no means for him to know what are the trace event types used in the instrumented library so he has some chance of duplicating some of them and thus to obtain a contaminated tracing of his application.

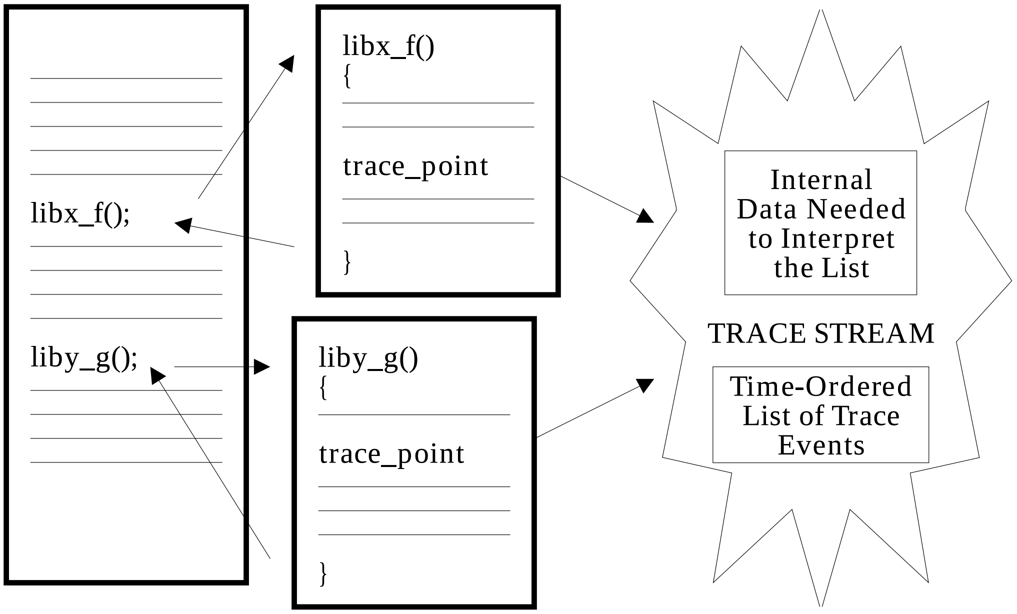


Figure B-6 Trace Name Space Overview: With Third-Party Library

We have requirements to allow program images containing pieces from various vendors to be traced without also requiring those or any other vendors to coordinate their uses of the trace facility, and especially the naming of their various trace event types and trace point IDs. The chosen solution is to provide a very large name space, large enough so that the individual vendors can give their trace types and tracepoint IDs sufficiently long and descriptive names making the occurrence of collisions quite unlikely. The probability of collision is thus made sufficiently low so that the problem may, as a practical matter, be ignored. By requirement, the consequence of collisions will be a slight ambiguity in the trace streams; tracing will continue in spite of collisions and ambiguities. ‘‘The show must go on’’. The *posix_prog_address* member of the **posix_trace_event_info** structure is used to allow trace streams to be unambiguously interpreted, despite the fact that trace event types and trace event names need not be unique.

The *posix_trace_eventid_open*( ) function is required to allow the instrumented third-party library to get a valid trace event type identifier for its trace event names. This operation is, somehow, an allocation, and the group was aware of proposing some deallocation mechanism which the instrumented application could use to recover the resources used by a trace event type identifier. This would have given the instrumented application the benefit of being capable of reusing a possible minimum set of trace event type identifiers, but also the inconvenience to have, possibly in the same trace stream, one trace event type identifier identifying two different trace event types. After some discussions the group decided to not define such a function which would make this API thicker for little benefit, the user having always the possibility of adding identification information in the data member of the trace event structure.

The set of the trace event type identifiers the controlling process wants to filter out is initialized in the trace mechanism using the function *posix_trace_set_filter*( ), setting the arguments according to the definitions explained in *posix_trace_set_filter*( ). This operation can be done statically (when the trace is in the STOPPED state) or dynamically (when the trace is in the STARTED state). The preparation of the filter is normally done using the function defined in *posix_trace_eventtypelist_getnext_id*( ) and eventually the function *posix_trace_eventtypelist_rewind*( ) in order to know (before the recording) the list of the potential set of trace event types that can be recorded. In the case of an active trace stream, this list may not be exhaustive. Actually, the target process may not have yet called the function *posix_trace_eventid_open*( ). But it is a common practice, for a controlling process, to prepare the filtering of a future trace stream before its start. Therefore the user must have a way to get the trace event type identifier corresponding to a well-known trace event name before its future association by the pre-cited function. This is done by calling the *posix_trace_trid_eventid_open*( ) function, given the trace stream identifier and the trace name, and described hereafter. Because this trace event type identifier is associated with a trace stream identifier, where a unique process has initialized two or more traces, the implementation is expected to return the same trace event type identifier for successive calls to *posix_trace_trid_eventid_open*( ) with different trace stream identifiers. The *posix_trace_eventid_get_name*( ) function is used by the controller process to identify, by the name, the trace event type returned by a call to the *posix_trace_eventtypelist_getnext_id*( ) function.

Afterwards, the set of trace event types is constructed using the functions defined in *posix_trace_eventset_empty*( ), *posix_trace_eventset_fill*( ), *posix_trace_eventset_add*( ), and *posix_trace_eventset_del*( ).

A set of functions is provided devoted to the manipulation of the trace event type identifier and names for an active trace stream. All these functions require the trace stream identifier argument as the first parameter. The opacity of the trace event type identifier implies that the user cannot associate directly its well-known trace event name with the system associated trace event type identifier.

The *posix_trace_trid_eventid_open*( ) function allows the application to get the system trace event type identifier back from the system, given its well-known trace event name. This function is useful only when a controlling process needs to specify specific events to be filtered. |

The *posix_trace_eventid_get_name*( ) function allows the application to obtain a trace event name given its trace event type identifier. One possible use of this function is to identify the type of a trace event retrieved from the trace stream, and print it. The easiest way to implement this requirement, is to use a single trace event type map for all the processes whose maps are required to be identical. A more difficult way is to attempt to keep multiple maps identical at every call to *posix_trace_eventid_open*( ) and *posix_trace_trid_eventid_open*( ).

B.2.11.6 Rationale on Trace Events Type Filtering

The most basic rationale for runtime and pre-registration filtering (selection/rejection) of trace event types is to prevent choking of the trace collection facility, and/or overloading of the computer system. Any worthwhile trace facility can bring even the largest computer to its knees. Otherwise, we would record everything, and filter after the fact; it would be much simpler, but impractical.

To achieve debugging, measurement, or whatever the purpose of tracing, the filtering of trace event types is an important part of trace analysis. Due to the fact that the trace events are put into a trace stream and probably logged afterwards into a file, different levels of filtering—that is, rejection of trace event types—are possible.

Filtering of Trace Event Types Before Tracing

This function, represented by the *posix_trace_set_filter*( ) function in IEEE Std 1003.1-200x (see *posix_trace_set_filter*( )), selects, before or during tracing, the set of trace event types to be filtered out. It should be possible also (as OSF suggested in their ETAP trace specifications) to select the kernel trace event types to be traced in a system-wide fashion. These two functionalities are called the pre-filtering of trace event types.

The restriction on the actual type used for the **trace_event_set_t** type is intended to guarantee that these objects can always be assigned, have their address taken, and be passed by value as parameters. It is not intended that this type be a structure including pointers to other data structures, as that could impact the portability of applications performing such operations. A reasonable implementation could be a structure containing an array of integer types.

Filtering of Trace Event Types at Runtime

Using this API, this functionality may be built, a privileged process or a privileged thread can get trace events from the trace stream of another process or thread, and thus specify the type of trace events to record into a file, using methods and interfaces out of the scope of IEEE Std 1003.1-200x. This functionality, called inline filtering of trace event types, is used for runtime analysis of trace streams.

Post-Mortem Filtering of Trace Event Types

The word *post-mortem* is used here to indicate that some unanticipated situation occurs during execution that does not permit a pre or inline filtering of trace events and that it is necessary to record all trace event types, to have a chance to discover the problem afterwards. When the program stops, all the trace events recorded previously can be analyzed in order to find the solution. This functionality could be named the post-filtering of trace event types.

Discussions about Trace Event Type-Filtering

After long discussions with the parties involved in the process of defining the trace interface, it seems that the sensitivity to the filtering problem is different, but everybody agrees that the level of the overhead introduced during the tracing operation depends on the filtering method elected. If the time that it takes the trace event to be recorded can be neglected, the overhead introduced by the filtering process can be classified as follows:

Pre-filtering	System and process / thread-level overhead
Inline-filtering	Process / thread-level overhead
Post-filtering	No overhead; done offline

The pre-filtering could be named *critical realtime* filtering in the sense that the filtering of trace event type is manageable at the user level so the user can lower to a minimum the filtering overhead at some user selected level of priority for the inline filtering, or delay the filtering to after execution for the post-filtering. The counterpart of this solution is that the size of the trace stream must be sufficient to record all the trace events. The advantage of the pre-filtering is that the utilization of the trace stream is optimized.

Only pre-filtering is defined by IEEE Std 1003.1-200x. However, great care must be taken in specifying pre-filtering, so that it does not impose unacceptable overhead. Moreover, it is necessary to isolate all the functionality relative to the pre-filtering.

The result of this rationale is to define a new option, the Trace Event Filter option, not necessarily implemented in small realtime systems, where system overhead is minimized to the extent possible.

B.2.11.7 *Tracing, pthread API*

The objective to be able to control tracing for individual threads may be in conflict with the efficiency expected in threads with a contentionscope attribute of PTHREAD_SCOPE_PROCESS. For these threads, context switches from one thread that has tracing enabled to another thread that has tracing disabled may require a kernel call to inform the kernel whether it has to trace system events executed by that thread or not. For this reason, it was proposed that the ability to enable or disable tracing for PTHREAD_SCOPE_PROCESS threads be made optional, through the introduction of a Trace Scope Process option. A trace implementation which did not implement the Trace Scope Process option would not honor the tracing-state attribute of a thread with PTHREAD_SCOPE_PROCESS; it would, however, honor the tracing-state attribute of a thread with PTHREAD_SCOPE_SYSTEM. This proposal was rejected as:

1. Removing desired functionality (per-thread trace control)
2. Introducing counter-intuitive behavior for the tracing-state attribute
3. Mixing logically orthogonal ideas (thread scheduling and thread tracing) [Objective 4]

Finally, to solve this complex issue, this API does not provide *pthread_gettracingstate*( ), *pthread_settracingstate*( ), *pthread_attr_gettracingstate*( ), and *pthread_attr_settracingstate*( ) interfaces. These interfaces force the thread implementation to add to the weight of the thread and cause a revision of the threads libraries, just to support tracing. Worse yet, *posix_trace_userevent*( ) must always test this per-thread variable even in the common case where it is not used at all. Per-thread tracing is easy to implement using existing interfaces where necessary; see the following example.

Example

```
/* Caution. Error checks omitted */
static pthread_key_t my_key;
static trace_event_id_t my_event_id;
static pthread_once_t my_once = PTHREAD_ONCE_INIT;

void my_init(void)
{
    (void) pthread_key_create(&my_key, NULL);
    (void) posix_trace_eventid_open("my", &my_event_id);
}

int get_trace_flag(void)
{
    pthread_once(&my_once, my_init);
    return (pthread_getspecific(my_key) != NULL);
}

void set_trace_flag(int f)
{
    pthread_once(&my_once, my_init);
    pthread_setspecific(my_key, f? &my_event_id: NULL);
}

fn()
{
    if (get_trace_flag())
```

```
        posix_trace_event(my_event_id, ...)
}
```

The above example does not implement third-party state setting, but it is also implementable with some more work, yet the extra functionality is rarely needed.

Lastly, per-thread tracing works poorly for threads with PTHREAD_SCOPE_PROCESS contention scope. These ‘‘library’’ threads have minimal interaction with the kernel and would have to explicitly set the attributes whenever they are context switched to a new kernel thread in order to trace system events. Such state was explicitly avoided in POSIX threads to keep PTHREAD_SCOPE_PROCESS threads lightweight.

The reason that keeping PTHREAD_SCOPE_PROCESS threads lightweight is important is that such threads can be used not just for simple multi-processors but also for coroutine style programming (such as discrete event simulation) without inventing a new threads paradigm. Adding extra runtime cost to thread context switches will make using POSIX threads less attractive in these situations.

B.2.11.8 Rationale on Triggering

The ability to start or stop tracing based on the occurrence of specific trace event types has been proposed as a parallel to similar functionality appearing in logic analyzers. Such triggering, in order to be very useful, should be based not only on the trace event type, but on trace event-specific data, including tests of user-specified fields for matching or threshold values.

Such a facility is unnecessary where the buffering of the stream is not a constraint, since such checks can be performed offline during post-mortem analysis.

For example, a large system could incorporate a daemon utility to collect the trace records from memory buffers and spool them to secondary storage for later analysis. In the instances where resources are truly limited, such as embedded applications, the application incorporation of application code to test the circumstances of a trace event and call the trace point only if needed is usually straightforward.

For performance reasons, the *posix_trace_event*( ) function should be implemented using a macro, so if the trace is inactive, the trace event point calls are latent code and must cost no more than a scalar test.

The API proposed in IEEE Std 1003.1-200x does not include any triggering functionality.

B.2.11.9 Rationale on Timestamp Clock

It has been suggested that the tracing mechanism should include the possibility of specifying the clock to be used in timestamping the trace events. When application trace events must be correlated to remote trace events, such a facility could provide a global time reference not available from a local clock. Further, the application may be driven by timers based on a clock different from that used for the timestamp, and the correlation of the trace to those untraced timer activities could be an important part of the analysis of the application.

However, the tracing mechanism needs to be fast and just the provision of such an option can materially affect its performance. Leaving aside the performance costs of reading some clocks, this notion is also ill-defined when kernel trace events are to be traced by two applications making use of different tracing clocks. This can even happen within a single application where different parts of the application are served by different clocks. Another complication can occur when a clock is maintained strictly at the user level and is unavailable at the kernel level.

It is felt that the benefits of a selectable trace clock do not match its costs. Applications that wish to correlate clocks other than the default tracing clock can include trace events with sample values of those other clocks, allowing correlation of timestamps from the various independent clocks. In any case, such a technique would be required when applications are sensitive to multiple clocks.

B.2.11.10 Rationale on Different Overrun Conditions

The analysis of the dynamic behavior of the trace mechanism shows that different overrun conditions may occur. The API must provide a means to manage such conditions in a portable way.

Overrun in Trace Streams Initialized with POSIX_TRACE_LOOP Policy

In this case, the user of the trace mechanism is interested in using the trace stream with POSIX_TRACE_LOOP policy to record trace events continuously, but ideally without losing any trace events. The online analyzer process must get the trace events at a mean speed equivalent to the recording speed. Should the trace stream become full, a trace stream overrun occurs. This condition is detected by getting the status of the active trace stream (function *posix_trace_get_status*( )) and looking at the member *posix_stream_overrun_status* of the read **posix_stream_status** structure. In addition, two predefined trace event types are defined:

1. The beginning of a trace overflow, to locate the beginning of an overflow when reading a trace stream
2. The end of a trace overflow, to locate the end of an overflow, when reading a trace stream

As a timestamp is associated with these predefined trace events, it is possible to know the duration of the overflow.

Overrun in Dumping Trace Streams into Trace Logs

The user lets the trace mechanism dump the trace stream initialized with POSIX_TRACE_FLUSH policy automatically into a trace log. If the dump operation is slower than the recording of trace events, the trace stream can overrun. This condition is detected by getting the status of the active trace stream (function *posix_trace_get_status*( )) and looking at the member *posix_log_overrun_status* of the read **posix_stream_status** structure. This overrun indicates that the trace mechanism is not able to operate in this mode at this speed. It is the responsibility of the user to modify one of the trace parameters (the stream size or the trace event type filter, for instance) to avoid such overrun conditions, if overruns are to be prevented. The same already predefined trace event types (see **Overrun in Trace Streams Initialized with POSIX_TRACE_LOOP Policy**) are used to detect and to know the duration of an overflow.

Reading an Active Trace Stream

Although this trace API allows one to read an active trace stream with log while it is tracing, this feature can lead to false overflow origin interpretation: the trace log or the reader of the trace stream. Reading from an active trace stream with log is thus non-portable, and has been left unspecified.

B.2.12 Data Types

The requirement that additional types defined in this section end in ‘‘_t’’ was prompted by the problem of name space pollution. It is difficult to define a type (where that type is not one defined by IEEE Std 1003.1-200x) in one header file and use it in another without adding symbols to the name space of the program. To allow implementors to provide their own types, all conforming applications are required to avoid symbols ending in ‘‘_t’’, which permits the implementor to provide additional types. Because a major use of types is in the definition of structure members, which can (and in many cases must) be added to the structures defined in IEEE Std 1003.1-200x, the need for additional types is compelling.

The types, such as **ushort** and **ulong**, which are in common usage, are not defined in IEEE Std 1003.1-200x (although **ushort_t** would be permitted as an extension). They can be added to **<sys/types.h>** using a feature test macro (see Section B.2.2.1 (on page 3375)). A suggested symbol for these is _SYSIII. Similarly, the types like **u_short** would probably be best controlled by _BSD.

Some of these symbols may appear in other headers; see Section B.2.2.2 (on page 3376).

dev_t	This type may be made large enough to accommodate host-locality considerations of networked systems. This type must be arithmetic. Earlier proposals allowed this to be non-arithmetic (such as a structure) and provided a *samefile*( ) function for comparison.
gid_t	Some implementations had separated **gid_t** from **uid_t** before POSIX.1 was completed. It would be difficult for them to coalesce them when it was unnecessary. Additionally, it is quite possible that user IDs might be different from group IDs because the user ID might wish to span a heterogeneous network, where the group ID might not. For current implementations, the cost of having a separate **gid_t** will be only lexical.
mode_t	This type was chosen so that implementations could choose the appropriate integral type, and for compatibility with the ISO C standard. 4.3 BSD uses **unsigned short** and the SVID uses **ushort**, which is the same. Historically, only the low-order sixteen bits are significant.
nlink_t	This type was introduced in place of **short** for *st_nlink* (see the **<sys/stat.h>** header) in response to an objection that **short** was too small.
off_t	This type is used only in *lseek*( ), *fcntl*( ), and **<sys/stat.h>**. Many implementations would have difficulties if it were defined as anything other than **long**. Requiring an integral type limits the capabilities of *lseek*( ) to four gigabytes. The ISO C standard supplies routines that use larger types; see *fgetpos*( ) and *fsetpos*( ). XSI-conformant systems provide the *fseeko*( ) and *lseeko*( ) functions that use larger types.
pid_t	The inclusion of this symbol was controversial because it is tied to the issue of the representation of a process ID as a number. From the point of view of a conforming application, process IDs should be ‘‘magic cookies’’[1] that are produced

---

[1] An historical term meaning: ‘‘An opaque object, or token, of determinate size, whose significance is known only to the entity which created it. An entity receiving such a token from the generating entity may only make such use of the ‘cookie’ as is defined and permitted by the supplying entity.’’

by calls such as *fork*( ), used by calls such as *waitpid*( ) or *kill*( ), and not otherwise analyzed (except that the sign is used as a flag for certain operations).

The concept of a {PID_MAX} value interacted with this in early proposals. Treating process IDs as an opaque type both removes the requirement for {PID_MAX} and allows systems to be more flexible in providing process IDs that span a large range of values, or a small one.

Since the values in **uid_t**, **gid_t**, and **pid_t** will be numbers generally, and potentially both large in magnitude and sparse, applications that are based on arrays of objects of this type are unlikely to be fully portable in any case. Solutions that treat them as magic cookies will be portable.

{CHILD_MAX} precludes the possibility of a ‘‘toy implementation’’, where there would only be one process.

ssize_t This is intended to be a signed analog of **size_t**. The wording is such that an implementation may either choose to use a longer type or simply to use the signed version of the type that underlies **size_t**. All functions that return **ssize_t** (*read*( ) and *write*( )) describe as ‘‘implementation-defined’’ the result of an input exceeding {SSIZE_MAX}. It is recognized that some implementations might have **int**s that are smaller than **size_t**. A conforming application would be constrained not to perform I/O in pieces larger than {SSIZE_MAX}, but a conforming application using extensions would be able to use the full range if the implementation provided an extended range, while still having a single type-compatible interface.

The symbols **size_t** and **ssize_t** are also required in **<unistd.h>** to minimize the changes needed for calls to *read*( ) and *write*( ). Implementors are reminded that it must be possible to include both **<sys/types.h>** and **<unistd.h>** in the same program (in either order) without error.

uid_t Before the addition of this type, the data types used to represent these values varied throughout early proposals. The **<sys/stat.h>** header defined these values as type **short**, the **<passwd.h>** file (now **<pwd.h>** and **<grp.h>**) used an **int**, and *getuid*( ) returned an **int**. In response to a strong objection to the inconsistent definitions, all the types to were switched to **uid_t**.

In practice, those historical implementations that use varying types of this sort can typedef **uid_t** to **short** with no serious consequences.

The problem associated with this change concerns object compatibility after structure size changes. Since most implementations will define **uid_t** as a short, the only substantive change will be a reduction in the size of the **passwd** structure. Consequently, implementations with an overriding concern for object compatibility can pad the structure back to its current size. For that reason, this problem was not considered critical enough to warrant the addition of a separate type to POSIX.1.

The types **uid_t** and **gid_t** are magic cookies. There is no {UID_MAX} defined by POSIX.1, and no structure imposed on **uid_t** and **gid_t** other than that they be positive arithmetic types. (In fact, they could be **unsigned char**.) There is no maximum or minimum specified for the number of distinct user or group IDs.

B.3 System Interfaces

See the RATIONALE sections on the individual reference pages.

B.3.1 Examples for Spawn

The following long examples are provided in the Rationale (Informative) volume of IEEE Std 1003.1-200x as a supplement to the reference page for *spawn*( ).

Example Library Implementation of Spawn

The *posix_spawn*( ) or *posix_spawnp*( ) functions provide the following:

- Simply start a process executing a process image. This is the simplest application for process creation, and it may cover most executions of POSIX *fork*( ).
- Support I/O redirection, including pipes.
- Run the child under a user and group ID in the domain of the parent.
- Run the child at any priority in the domain of the parent.

The *posix_spawn*( ) or *posix_spawnp*( ) functions do not cover every possible use of the *fork*( ) function, but they do span the common applications: typical use by a shell and a login utility.

The price for an application is that before it calls *posix_spawn*( ) or *posix_spawnp*( ), the parent must adjust to a state that *posix_spawn*( ) or *posix_spawnp*( ) can map to the desired state for the child. Environment changes require the parent to save some of its state and restore it afterwards. The effective behavior of a successful invocation of *posix_spawn*( ) is as if the operation were implemented with POSIX operations as follows:

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sched.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <signal.h>

/* #include <spawn.h>*/
/*******************************************/
/* Things that could be defined in spawn.h */
/*******************************************/
typedef struct
    {
    short posix_attr_flags;
#define POSIX_SPAWN_SETPGROUP       0x1
#define POSIX_SPAWN_SETSIGMASK      0x2
#define POSIX_SPAWN_SETSIGDEF       0x4
#define POSIX_SPAWN_SETSCHEDULER    0x8
#define POSIX_SPAWN_SETSCHEDPARAM   0x10
#define POSIX_SPAWN_RESETIDS        0x20
    pid_t posix_attr_pgroup;
    sigset_t posix_attr_sigmask;
    sigset_t posix_attr_sigdefault;
```

```
    int posix_attr_schedpolicy;
    struct sched_param posix_attr_schedparam;
    } posix_spawnattr_t;

typedef char *posix_spawn_file_actions_t;

int posix_spawn_file_actions_init(
        posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_destroy(
        posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
        posix_spawn_file_actions_t *file_actions, int fildes);
int posix_spawn_file_actions_adddup2(
        posix_spawn_file_actions_t *file_actions, int fildes,
        int newfildes);
int posix_spawn_file_actions_addopen(
        posix_spawn_file_actions_t *file_actions, int fildes,
        const char *path, int oflag, mode_t mode);
int posix_spawnattr_init(posix_spawnattr_t *attr);
int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_getflags(const posix_spawnattr_t *attr, short *lags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
int posix_spawnattr_getpgroup(const posix_spawnattr_t *attr,
        pid_t *pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr,
        int *schedpolicy);
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
        int schedpolicy);
int posix_spawnattr_getschedparam(const posix_spawnattr_t *attr,
        struct sched_param *schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *attr,
        const struct sched_param *schedparam);
int posix_spawnattr_getsigmask(const posix_spawnattr_t *attr,
        sigset_t *sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
        const sigset_t *sigmask);
int posix_spawnattr_getdefault(const posix_spawnattr_t *attr,
        sigset_t *sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
        const sigset_t *sigdefault);
int posix_spawn(pid_t *pid, const char *path,
        const posix_spawn_file_actions_t *file_actions,
        const posix_spawnattr_t *attrp, char * const argv[],
        char * const envp[]);
int posix_spawnp(pid_t *pid, const char *file,
        const posix_spawn_file_actions_t *file_actions,
        const posix_spawnattr_t *attrp, char * const argv[],
        char * const envp[]);

/*****************************************/
/* Example posix_spawn() library routine */
/*****************************************/
int posix_spawn(pid_t *pid,
```

```
            const char *path,
            const posix_spawn_file_actions_t *file_actions,
            const posix_spawnattr_t *attrp,
            char * const argv[],
            char * const envp[])
           {
           /* Create process */
           if((*pid=fork()) == (pid_t)0)
               {
               /* This is the child process */
               /* Worry about process group */
               if(attrp->posix_attr_flags & POSIX_SPAWN_SETPGROUP)
                   {
                   /* Override inherited process group */
                   if(setpgid(0, attrp->posix_attr_pgroup) != 0)
                       {
                       /* Failed */
                       exit(127);
                       }
                   }

                /* Worry about process signal mask */
                if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGMASK)
                    {
                    /* Set the signal mask (can't fail) */
                    sigprocmask(SIG_SETMASK , &attrp->posix_attr_sigmask,
                        NULL);
                    }

                /* Worry about resetting effective user and group IDs */
                if(attrp->posix_attr_flags & POSIX_SPAWN_RESETIDS)
                    {
                    /* None of these can fail for this case. */
                    setuid(getuid());
                    setgid(getgid());
                    }

                /* Worry about defaulted signals */
                if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGDEF)
                    {
                    struct sigaction deflt;
                    sigset_t all_signals;

                    int s;

                    /* Construct default signal action */
                    deflt.sa_handler = SIG_DFL;
                    deflt.sa_flags = 0;

                    /* Construct the set of all signals */
                    sigfillset(&all_signals);

                    /* Loop for all signals */
                    for(s=0; sigismember(&all_signals,s); s++)
                        {
                        /* Signal to be defaulted? */
```

```
                if(sigismember(&attrp->posix_attr_sigdefault,s))
                    {
                    /* Yes; default this signal */
                    if(sigaction(s, &deflt, NULL) == -1)
                        {
                        /* Failed */
                        exit(127);
                        }
                    }
                }
            }

        /* Worry about the fds if we are to map them */
        if(file_actions != NULL)
            {
            /* Loop for all actions in object file_actions */
            /*(implementation dives beneath abstraction)*/
            char *p = *file_actions;
            while(*p != ’ ’)
                {
                if(strncmp(p,"close(",6) == 0)
                    {
                    int fd;
                    if(sscanf(p+6,"%d)",&fd) != 1)
                        {
                        exit(127);
                        }
                    if(close(fd) == -1) exit(127);
                    }
                else if(strncmp(p,"dup2(",5) == 0)
                    {
                    int fd,newfd;
                    if(sscanf(p+5,"%d,%d)",&fd,&newfd) != 2)
                        {
                        exit(127);
                        }
                    if(dup2(fd, newfd) == -1) exit(127);
                    }
                else if(strncmp(p,"open(",5) == 0)
                    {
                    int fd,oflag;
                    mode_t mode;
                    int tempfd;
                    char path[1000]; /* Should be dynamic */
                    char *q;
                    if(sscanf(p+5,"%d,",&fd) != 1)
                        {
                         exit(127);
                        }
                    p = strchr(p, ’,’) + 1;
                    q = strchr(p, ’*’);
                    if(q == NULL) exit(127);
                    strncpy(path, p, q-p);
```

```
                        path[q-p] = ’ ’;
                    if(sscanf(q+1,"%o,%o)",&oflag,&mode)!=2)
                        {
                        exit(127);
                        }
                    if(close(fd) == -1)
                        {
                        if(errno != EBADF) exit(127);
                        }
                    tempfd = open(path, oflag, mode);
                    if(tempfd == -1) exit(127);
                    if(tempfd != fd)
                        {
                        if(dup2(tempfd,fd) == -1)
                            {
                            exit(127);
                            }
                        if(close(tempfd) == -1)
                            {
                            exit(127);
                            }
                        }
                    }
                    else
                    {
                    exit(127);
                    }
                p = strchr(p, ’)’) + 1;
                }
            }

            /* Worry about setting new scheduling policy and parameters */
            if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDULER)
                {
                if(sched_setscheduler(0, attrp->posix_attr_schedpolicy,
                    &attrp->posix_attr_schedparam) == -1)
                    {
                    exit(127);
                    }
                }

            /* Worry about setting only new scheduling parameters */
            if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDPARAM)
                {
                if(sched_setparam(0, &attrp->posix_attr_schedparam)==-1)
                    {
                    exit(127);
                    }
                }

            /* Now execute the program at path */
            /* Any fd that still has FD_CLOEXEC set will be closed */
            execve(path, argv, envp);
            exit(127); /* exec failed */
```

```
                }
                else
                {
                /* This is the parent (calling) process */
                if(*pid == (pid_t)-1) return errno;
                return 0;
                }
            }

        /*****************************************************/
        /* Here is a crude but effective implementation of the */
        /* file action object operators which store actions as */
        /* concatenated token separated strings.               */
        /*****************************************************/
        /* Create object with no actions. */
        int posix_spawn_file_actions_init(
                posix_spawn_file_actions_t *file_actions)
            {
            *file_actions = malloc(sizeof(char));
            if(*file_actions == NULL) return ENOMEM;
            strcpy(*file_actions, "");
            return 0;
            }

        /* Free object storage and make invalid. */
        int posix_spawn_file_actions_destroy(
                posix_spawn_file_actions_t *file_actions)
            {
            free(*file_actions);
            *file_actions = NULL;
            return 0;
            }

        /* Add a new action string to object. */
        static int add_to_file_actions(
                posix_spawn_file_actions_t *file_actions,
                    char *new_action)
            {
            *file_actions = realloc
                (*file_actions, strlen(*file_actions)+strlen(new_action)+1);
            if(*file_actions == NULL) return ENOMEM;
            strcat(*file_actions, new_action);
            return 0;
            }

        /* Add a close action to object. */
        int posix_spawn_file_actions_addclose(
                posix_spawn_file_actions_t *file_actions, int fildes)
            {
            char temp[100];
            sprintf(temp, "close(%d)", fildes);
            return add_to_file_actions(file_actions, temp);
            }
```

```
/* Add a dup2 action to object. */
int posix_spawn_file_actions_adddup2(
        posix_spawn_file_actions_t *file_actions, int fildes,
        int newfildes)
    {
    char temp[100];
    sprintf(temp, "dup2(%d,%d)", fildes, newfildes);
    return add_to_file_actions(file_actions, temp);
    }

/* Add an open action to object. */
int posix_spawn_file_actions_addopen(
        posix_spawn_file_actions_t *file_actions, int fildes,
        const char *path, int oflag, mode_t mode)
    {
    char temp[100];
    sprintf(temp, "open(%d,%s*%o,%o)", fildes, path, oflag, mode);
    return add_to_file_actions(file_actions, temp);
    }

/*******************************************************/
/* Here is a crude but effective implementation of the */
/* spawn attributes object functions which manipulate  */
/* the individual attributes.                          */
/*******************************************************/
/* Initialize object with default values. */
int posix_spawnattr_init(
        posix_spawnattr_t *attr)
    {
    attr->posix_attr_flags=0;
    attr->posix_attr_pgroup=0;
    /* Default value of signal mask is the parent’s signal mask; */
    /* other values are also allowed */
    sigprocmask(0,NULL,&attr->posix_attr_sigmask);
    sigemptyset(&attr->posix_attr_sigdefault);
    /* Default values of scheduling attr inherited from the parent; */
    /* other values are also allowed */
    attr->posix_attr_schedpolicy=sched_getscheduler(0);
    sched_getparam(0,&attr->posix_attr_schedparam);
    return 0;
    }

int posix_spawnattr_destroy(posix_spawnattr_t *attr)
    {
    /* No action needed */
    return 0;
    }

int posix_spawnattr_getflags(const posix_spawnattr_t *attr,
        short *flags)
    {
    *flags=attr->posix_attr_flags;
    return 0;
    }
```

```
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags)
    {
    attr->posix_attr_flags=flags;
    return 0;
    }

int posix_spawnattr_getpgroup(const posix_spawnattr_t *attr,
        pid_t *pgroup)
    {
    *pgroup=attr->posix_attr_pgroup;
    return 0;
    }

int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup)
    {
    attr->posix_attr_pgroup=pgroup;
    return 0;
    }

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *attr,
        int *schedpolicy)
    {
    *schedpolicy=attr->posix_attr_schedpolicy;
    return 0;
    }

int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
        int schedpolicy)
    {
    attr->posix_attr_schedpolicy=schedpolicy;
    return 0;
    }

int posix_spawnattr_getschedparam(const posix_spawnattr_t *attr,
        struct sched_param *schedparam)
    {
    *schedparam=attr->posix_attr_schedparam;
    return 0;
    }

int posix_spawnattr_setschedparam(posix_spawnattr_t *attr,
        const struct sched_param *schedparam)
    {
    attr->posix_attr_schedparam=*schedparam;
    return 0;
    }

int posix_spawnattr_getsigmask(const posix_spawnattr_t *attr,
        sigset_t *sigmask)
    {
    *sigmask=attr->posix_attr_sigmask;
    return 0;
    }

int posix_spawnattr_setsigmask(posix_spawnattr_t *attr,
        const sigset_t *sigmask)
```

```
        {
        attr->posix_attr_sigmask=*sigmask;
        return 0;
        }
    int posix_spawnattr_getsigdefault(const posix_spawnattr_t *attr,
            sigset_t *sigdefault)
        {
        *sigdefault=attr->posix_attr_sigdefault;
        return 0;
        }
    int posix_spawnattr_setsigdefault(posix_spawnattr_t *attr,
            const sigset_t *sigdefault)
        {
        attr->posix_attr_sigdefault=*sigdefault;
        return 0;
        }
```

I/O Redirection with Spawn

I/O redirection with *posix_spawn*( ) or *posix_spawnp*( ) is accomplished by crafting a *file_actions* argument to effect the desired redirection. Such a redirection follows the general outline of the following example:

```
/* To redirect new standard output (fd 1) to a file, */
/* and redirect new standard input (fd 0) from my fd socket_pair[1], */
/* and close my fd socket_pair[0] in the new process. */
posix_spawn_file_actions_t file_actions;
posix_spawn_file_actions_init(&file_actions);
posix_spawn_file_actions_addopen(&file_actions, 1, "newout", ...);
posix_spawn_file_actions_dup2(&file_actions, socket_pair[1], 0);
posix_spawn_file_actions_close(&file_actions, socket_pair[0]);
posix_spawn_file_actions_close(&file_actions, socket_pair[1]);
posix_spawn(..., &file_actions, ...);
posix_spawn_file_actions_destroy(&file_actions);
```

Spawning a Process Under a New User ID

Spawning a process under a new user ID follows the outline shown in the following example:

```
Save = getuid();
setuid(newid);
posix_spawn(...);
setuid(Save);
```

Rationale (Informative)

Part C:

Shell and Utilities

The Open Group
The Institute of Electrical and Electronics Engineers, Inc.

Appendix C

Rationale for Shell and Utilities

C.1 Introduction

C.1.1 Scope

Refer to Section A.1.1 (on page 3293).

C.1.2 Conformance

Refer to Section A.2 (on page 3299).

C.1.3 Normative References

There is no additional rationale provided for this section.

C.1.4 Change History

The change history is provided as an informative section, to track changes from previous issues of IEEE Std 1003.1-200x.

The following sections describe changes made to the Shell and Utilities volume of IEEE Std 1003.1-200x since Issue 5 of the base document. The CHANGE HISTORY section for each utility describes technical changes made to that utility since Issue 5. Changes between earlier issues of the base document and Issue 5 are not included.

The change history between Issue 5 and Issue 6 also lists the changes since the ISO POSIX-2: 1993 standard.

Changes from Issue 5 to Issue 6 (IEEE Std 1003.1-200x)

The following list summarizes the major changes that were made in the Shell and Utilities volume of IEEE Std 1003.1-200x from Issue 5 to Issue 6:

- This volume of IEEE Std 1003.1-200x is extensively revised so it can be both an IEEE POSIX Standard and an Open Group Technical Standard.
- The terminology has been reworked to meet the style requirements.
- Shading notation and margin codes are introduced for identification of options within the volume.
- This volume of IEEE Std 1003.1-200x is updated to mandate support of FIPS 151-2. The following changes were made:
    - Support is mandated for the capabilities associated with the following symbolic constants:

      _POSIX_CHOWN_RESTRICTED
      _POSIX_JOB_CONTROL
      _POSIX_SAVED_IDS

    - In the environment for the login shell, the environment variables *LOGNAME* and *HOME* shall be defined and have the properties described in the Base Definitions volume of

IEEE Std 1003.1-200x, Chapter 7, Locale.

- This volume of IEEE Std 1003.1-200x is updated to align with some features of the Single UNIX Specification.
- A new section on Utility Limits is added.
- A section on the Relationships to Other Documents is added.
- Concepts and definitions have been moved to a separate volume.
- A RATIONALE section is added to each reference page.
- The *c99* utility is added as a replacement for *c89*, which is withdrawn in this issue.
- IEEE Std 1003.2d-1994 is incorporated, adding the *qalter*, *qdel*, *qhold*, *qmove*, *qmsg*, *qrerun*, *qrls*, *qselect*, *qsig*, *qstat*, and *qsub* utilities.
- IEEE P1003.2b draft standard is incorporated, making extensive updates and adding the *iconv* utility.
- IEEE PASC Interpretations are applied.
- The Open Groups corrigenda and resolutions are applied.

New Features in Issue 6

The following table lists the new utilities introduced since the ISO POSIX-2:1993 standard (as modified by IEEE Std 1003.2d-1994). These are all part of the XSI extension.

New Utilities in Issue 6

admin	*fuser*	*link*	*tsort*	*uustat*
cal	*gencat*	*m4*	*ulimit*	*uux*
cflow	*get*	*nl*	*uncompress*	*val*
compress	*hash*	*prs*	*unget*	*what*
cxref	*ipcrm*	*sact*	*unlink*	*zcat*
delta	*ipcs*	*sccs*	*uucp*	

C.1.5 Terminology

Refer to Section A.1.4 (on page 3295).

C.1.6 Definitions

Refer to Section A.3 (on page 3302).

C.1.7 Relationship to Other Documents

C.1.7.1 System Interfaces

It has been pointed out that the Shell and Utilities volume of IEEE Std 1003.1-200x assumes that a great deal of functionality from the System Interfaces volume of IEEE Std 1003.1-200x is present, but never states exactly how much (and strictly does not need to since both are mandated on a conforming system). This section is an attempt to clarify the assumptions.

C.1.7.2 Concepts Derived from the ISO C Standard

This section was introduced to address the issue that there was insufficient detail presented by such utilities as *awk* or *sh* about their procedural control statements and their methods of performing arithmetic functions.

The ISO C standard was selected as a model because most historical implementations of the standard utilities were written in C. Thus, it was more likely that they would act in the desired manner without modification.

Using the ISO C standard is primarily a notational convenience so that the many procedural languages in the Shell and Utilities volume of IEEE Std 1003.1-200x would not have to be rigorously described in every aspect. Its selection does not require that the standard utilities be written in Standard C; they could be written in Common Usage C, Ada, Pascal, assembler language, or anything else.

The sizes of the various numeric values refer to C-language data types that are allowed to be different sizes by the ISO C standard. Thus, like a C-language application, a shell application cannot rely on their exact size. However, it can rely on their minimum sizes expressed in the ISO C standard, such as {LONG_MAX} for a **long** type.

C.1.8 Portability

Refer to Section A.1.5 (on page 3298).

C.1.8.1 Codes

Refer to Section A.1.5.1 (on page 3298).

C.1.9 Utility Limits

This section grew out of an idea that originated with the original POSIX.1, in the tables of system limits for the *sysconf*( ) and *pathconf*( ) functions. The idea being that a conforming application can be written to use the most restrictive values that a minimal system can provide, but it should not have to. The values provided represent compromises so that some vendors can use historically limited versions of UNIX system utilities. They are the highest values that a strictly conforming application can assume, given no other information.

However, by using the *getconf* utility or the *sysconf*( ) function, the elegant application can be tailored to more liberal values on some of the specific instances of specific implementations.

There is no explicitly stated requirement that an implementation provide finite limits for any of these numeric values; the implementation is free to provide essentially unbounded capabilities (where it makes sense), stopping only at reasonable points such as {ULONG_MAX} (from the ISO C standard). Therefore, applications desiring to tailor themselves to the values on a particular implementation need to be ready for possibly huge values; it may not be a good idea to allocate blindly a buffer for an input line based on the value of {LINE_MAX}, for instance. However, unlike the System Interfaces volume of IEEE Std 1003.1-200x, there is no set of limits that return a special indication meaning ''unbounded''. The implementation should always return an actual number, even if the number is very large.

The statement:

> ''It is not guaranteed that the application ...''

is an indication that many of these limits are designed to ensure that implementors design their utilities without arbitrary constraints related to unimaginative programming. There are certainly conditions under which combinations of options can cause failures that would not render an implementation non-conforming. For example, {EXPR_NEST_MAX} and {ARG_MAX} could collide when expressions are large; combinations of {BC_SCALE_MAX} and {BC_DIM_MAX} could exceed virtual memory.

In the Shell and Utilities volume of IEEE Std 1003.1-200x, the notion of a limit being guaranteed for the process lifetime, as it is in the System Interfaces volume of IEEE Std 1003.1-200x, is not as useful to a shell script. The *getconf* utility is probably a process itself, so the guarantee would be without value. Therefore, the Shell and Utilities volume of IEEE Std 1003.1-200x requires the guarantee to be for the session lifetime. This will mean that many vendors will either return very conservative values or possibly implement *getconf* as a built-in.

It may seem confusing to have limits that apply only to a single utility grouped into one global section. However, the alternative, which would be to disperse them out into their utility description sections, would cause great difficulty when *sysconf*( ) and *getconf* were described. Therefore, the standard developers chose the global approach.

Each language binding could provide symbol names that are slightly different from those shown here. For example, the C-Language Binding option adds a leading underscore to the symbols as a prefix.

The following comments describe selection criteria for the symbols and their values:

{ARG_MAX}
This is defined by the System Interfaces volume of IEEE Std 1003.1-200x. Unfortunately, it is very difficult for a conforming application to deal with this value, as it does not know how much of its argument space is being consumed by the environment variables of the user.

{BC_BASE_MAX}
{BC_DIM_MAX}
{BC_SCALE_MAX}
These were originally one value, {BC_SCALE_MAX}, but it was unreasonable to link all three concepts into one limit.

{CHILD_MAX}
This is defined by the System Interfaces volume of IEEE Std 1003.1-200x.

{COLL_WEIGHTS_MAX}
The weights assigned to **order** can be considered as ‘‘passes’’ through the collation algorithm.

{EXPR_NEST_MAX}
The value for expression nesting was borrowed from the ISO C standard.

{LINE_MAX}
This is a global limit that affects all utilities, unless otherwise noted. The {MAX_CANON} value from the System Interfaces volume of IEEE Std 1003.1-200x may further limit input lines from terminals. The {LINE_MAX} value was the subject of much debate and is a compromise between those who wished to have unlimited lines and those who understood that many historical utilities were written with fixed buffers. Frequently, utility writers selected the UNIX system constant BUFSIZ to allocate these buffers; therefore, some utilities were limited to 512 bytes for I/O lines, while others achieved 4 096 bytes or greater.

It should be noted that {LINE_MAX} applies only to input line length; there is no requirement in IEEE Std 1003.1-200x that limits the length of output lines. Utilities such as *awk*, *sed*, and *paste* could theoretically construct lines longer than any of the input lines they received, depending on the options used or the instructions from the application. They are not required to truncate their output to {LINE_MAX}. It is the responsibility of the application to deal with this. If the output of one of those utilities is to be piped into another of the standard utilities, line length restrictions will have to be considered; the *fold* utility, among others, could be used to ensure that only reasonable line lengths reach utilities or applications.

{LINK_MAX}
This is defined by the System Interfaces volume of IEEE Std 1003.1-200x.

{MAX_CANON}
{MAX_INPUT}
{NAME_MAX}
{NGROUPS_MAX}
{OPEN_MAX}
{PATH_MAX}
{PIPE_BUF}
These limits are defined by the System Interfaces volume of IEEE Std 1003.1-200x. Note that the byte lengths described by some of these values continue to represent bytes, even if the applicable character set uses a multi-byte encoding.

{RE_DUP_MAX}
The value selected is consistent with historical practice. Although the name implies that it applies to all REs, only BREs use the interval notation **\{***m*,*n***\}** addressed by this limit.

{POSIX2_SYMLINKS}
The {POSIX2_SYMLINKS} variable indicates that the underlying operating system supports the creation of symbolic links in specific directories. Many of the utilities defined in IEEE Std 1003.1-200x that deal with symbolic links do not depend on this value. For example, a utility that follows symbolic links (or does not, as the case may be) will only be affected by a symbolic link if it encounters one. Presumably, a file system that does not support symbolic links will not contain any. This variable does affect such utilities as *ln* −**s** and *pax* that attempt to create symbolic links.

{POSIX2_SYMLINKS} was developed even though there is no comparable configuration value for the system interfaces.

There are different limits associated with command lines and input to utilities, depending on the method of invocation. In the case of a C program *exec*-ing a utility, {ARG_MAX} is the underlying limit. In the case of the shell reading a script and *exec*-ing a utility, {LINE_MAX} limits the length of lines the shell is required to process, and {ARG_MAX} will still be a limit. If a user is entering a command on a terminal to the shell, requesting that it invoke the utility, {MAX_INPUT} may restrict the length of the line that can be given to the shell to a value below {LINE_MAX}.

When an option is supported, *getconf* returns a value of 1. For example, when C development is supported:

```
if [ "$(getconf POSIX2_C_DEV)" −eq 1 ]; then
    echo C supported
fi
```

The *sysconf*( ) function in the C-Language Binding option would return 1.

The following comments describe selection criteria for the symbols and their values:

POSIX2_C_BIND
POSIX2_C_DEV
POSIX2_FORT_DEV
POSIX2_FORT_RUN

POSIX2_SW_DEV
POSIX2_UPE

It is possible for some (usually privileged) operations to remove utilities that support these options or otherwise to render these options unsupported. The header files, the *sysconf*( ) function, or the *getconf* utility will not necessarily detect such actions, in which case they should not be considered as rendering the implementation non-conforming. A test suite should not attempt tests such as:

```
rm /usr/bin/c99
getconf POSIX2_C_DEV
```

POSIX2_LOCALEDEF

This symbol was introduced to allow implementations to restrict supported locales to only those supplied by the implementation.

C.1.10 Grammar Conventions

There is no additional rationale provided for this section.

C.1.11 Utility Description Defaults

This section is arranged with headings in the same order as all the utility descriptions. It is a collection of related and unrelated information concerning

1. The default actions of utilities
2. The meanings of notations used in IEEE Std 1003.1-200x that are specific to individual utility sections

Although this material may seem out of place here, it is important that this information appear before any of the utilities to be described later.

NAME

There is no additional rationale provided for this section.

SYNOPSIS

There is no additional rationale provided for this section.

DESCRIPTION

There is no additional rationale provided for this section.

OPTIONS

Although it has not always been possible, the standard developers tried to avoid repeating information to reduce the risk that duplicate explanations could each be modified differently.

The need to recognize −− is required because conforming applications need to shield their operands from any arbitrary options that the implementation may provide as an extension. For example, if the standard utility *foo* is listed as taking no options, and the application needed to give it a pathname with a leading hyphen, it could safely do it as:

```
foo −− −myfile
```

and avoid any problems with −**m** used as an extension.

OPERANDS

The usage of – is never shown in the SYNOPSIS. Similarly, the usage of – – is never shown.

The requirement for processing operands in command-line order is to avoid a ‘‘WeirdNIX’’ utility that might choose to sort the input files alphabetically, by size, or by directory order. Although this might be acceptable for some utilities, in general the programmer has a right to know exactly what order will be chosen.

Some of the standard utilities take multiple *file* operands and act as if they were processing the concatenation of those files. For example:

```
asa file1 file2
```

and:

```
cat file1 file2 | asa
```

have similar results when questions of file access, errors, and performance are ignored. Other utilities such as *grep* or *wc* have completely different results in these two cases. This latter type of utility is always identified in its DESCRIPTION or OPERANDS sections, whereas the former is not. Although it might be possible to create a general assertion about the former case, the following points must be addressed:

- Access times for the files might be different in the operand case *versus* the *cat* case.
- The utility may have error messages that are cognizant of the input filename, and this added value should not be suppressed. (As an example, *awk* sets a variable with the filename at each file boundary.)

STDIN

There is no additional rationale provided for this section.

INPUT FILES

A conforming application cannot assume the following three commands are equivalent:

```
tail –n +2 file
(sed –n 1q; cat) < file
cat file | (sed –n 1q; cat)
```

The second command is equivalent to the first only when the file is seekable. In the third command, if the file offset in the open file description were not unspecified, *sed* would have to be implemented so that it read from the pipe 1 byte at a time or it would have to employ some method to seek backwards on the pipe. Such functionality is not defined currently in POSIX.1 and does not exist on all historical systems. Other utilities, such as *head*, *read*, and *sh*, have similar properties, so the restriction is described globally in this section.

The definition of *text file* is strictly enforced for input to the standard utilities; very few of them list exceptions to the undefined results called for here. (Of course, ‘‘undefined’’ here does not mean that historical implementations necessarily have to change to start indicating error conditions. Conforming applications cannot rely on implementations succeeding or failing when non-text files are used.)

The utilities that allow line continuation are generally those that accept input languages, rather than pure data. It would be unusual for an input line of this type to exceed {LINE_MAX} bytes and unreasonable to require that the implementation allow unlimited accumulation of multiple lines, each of which could reach {LINE_MAX}. Thus, for a conforming application the total of all the continued lines in a set cannot exceed {LINE_MAX}.

The format description is intended to be sufficiently rigorous to allow other applications to generate these input files. However, since <blank>s can legitimately be included in some of the fields described by the standard utilities, particularly in locales other than the POSIX locale, this intent is not always realized.

ENVIRONMENT VARIABLES

There is no additional rationale provided for this section.

ASYNCHRONOUS EVENTS

Because there is no language prohibiting it, a utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process), and resignal itself.

STDOUT

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an *awk* or *lex* parser.

STDERR

This section does not describe error messages that refer to incorrect operation of the utility. Consider a utility that processes program source code as its input. This section is used to describe messages produced by a correctly operating utility that encounters an error in the program source code on which it is processing. However, a message indicating that the utility had insufficient memory in which to operate would not be described.

Some utilities have traditionally produced warning messages without returning a non-zero exit status; these are specifically noted in their sections. Other utilities shall not write to standard error if they complete successfully, unless the implementation provides some sort of extension to increase the verbosity or debugging level.

The format descriptions are intended to be sufficiently rigorous to allow post-processing of output by other programs.

OUTPUT FILES

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an *awk* or *lex* parser.

Receipt of the SIGQUIT signal should generally cause termination (unless in some debugging mode) that would bypass any attempted recovery actions.

EXTENDED DESCRIPTION

There is no additional rationale provided for this section.

EXIT STATUS

Note the additional discussion of exit values in *Exit Status for Commands* in the *sh* utility. It describes requirements for returning exit values greater than 125.

A utility may list zero as a successful return, 1 as a failure for a specific reason, and greater than 1 as ‘‘an error occurred’’. In this case, unspecified conditions may cause a 2 or 3, or other value, to be returned. A strictly conforming application should be written so that it tests for successful exit status values (zero in this case), rather than relying upon the single specific error value listed in IEEE Std 1003.1-200x. In that way, it will have maximum portability, even on implementations with extensions.

The standard developers are aware that the general non-enumeration of errors makes it difficult to write test suites that test the *incorrect* operation of utilities. There are some historical implementations that have expended effort to provide detailed status messages and a helpful environment to bypass or explain errors, such as prompting, retrying, or ignoring unimportant syntax errors; other implementations have not. Since there is no realistic way to mandate system behavior in cases of undefined application actions or system problems—in a manner acceptable to all cultures and environments—attention has been limited to the correct operation of utilities by the conforming application. Furthermore, the conforming application does not need detailed information concerning errors that it caused through incorrect usage or that it cannot correct.

There is no description of defaults for this section because all of the standard utilities specify something (or explicitly state ''Unspecified'') for exit status.

CONSEQUENCES OF ERRORS

Several actions are possible when a utility encounters an error condition, depending on the severity of the error and the state of the utility. Included in the possible actions of various utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; and validity checking of the file system or directory.

The text about recursive traversing is meant to ensure that utilities such as *find* process as many files in the hierarchy as they can. They should not abandon all of the hierarchy at the first error and resume with the next command-line operand, but should attempt to keep going.

APPLICATION USAGE

This section provides additional caveats, issues, and recommendations to the developer.

EXAMPLES

This section provides sample usage.

RATIONALE

There is no additional rationale provided for this section.

FUTURE DIRECTIONS

FUTURE DIRECTIONS sections act as pointers to related work that may impact the interface in the future, and often cautions the developer to architect the code to account for a change in this area. Note that a future directions statement should not be taken as a commitment to adopt a feature or interface in the future.

SEE ALSO

There is no additional rationale provided for this section.

CHANGE HISTORY

There is no additional rationale provided for this section.

C.1.12 Considerations for Utilities in Support of Files of Arbitrary Size

This section is intended to clarify the requirements for utilities in support of large files.

The utilities listed in this section are utilities which are used to perform administrative tasks such as to create, move, copy, remove, change the permissions, or measure the resources of a file. They are useful both as end-user tools and as utilities invoked by applications during software installation and operation.

The *chgrp*, *chmod*, *chown*, *ln*, and *rm* utilities probably require use of large file capable versions of *stat*( ), *lstat*( ), *ftw*( ), and the **stat** structure.

The *cat*, *cksum*, *cmp*, *cp*, *dd*, *mv*, *sum*, and *touch* utilities probably require use of large file capable versions of *creat*( ), *open*( ), and *fopen*( ).

The *cat*, *cksum*, *cmp*, *dd*, *df*, *du*, *ls*, and *sum* utilities may require writing large integer values. For example:

- The *cat* utility might have a **−n** option which counts <newline>s.
- The *cksum* and *ls* utilities report file sizes.
- The *cmp* utility reports the line number at which the first difference occurs, and also has a **−l** option which reports file offsets.
- The *dd*, *df*, *du*, *ls*, and *sum* utilities report block counts.

The *dd*, *find*, and *test* utilities may need to interpret command arguments that contain 64-bit values. For *dd*, the arguments include *skip*=*n*, *seek*=*n*, and *count*=*n*. For *find*, the arguments include **−size***n*. For *test*, the arguments are those associated with algebraic comparisons.

The *df* utility might need to access large file systems with *statvfs*( ).

The *ulimit* utility will need to use large file capable versions of *getrlimit*( ) and *setrlimit*( ) and be able to read and write large integer values.

C.1.13 Built-In Utilities

All of these utilities can be *exec*-ed. There is no requirement that these utilities are actually built into the shell itself, but many shells need the capability to do so because the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9.1.1, Command Search and Execution requires that they be found prior to the *PATH* search. The shell could satisfy its requirements by keeping a list of the names and directly accessing the file-system versions regardless of *PATH*. Providing all of the required functionality for those such as *cd* or *read* would be more difficult.

There were originally three justifications for allowing the omission of *exec*-able versions:

1. It would require wasting space in the file system, at the expense of very small systems. However, it has been pointed out that all 16 utilities in the table can be provided with 16 links to a single-line shell script:

   ```
   $0 "$@"
   ```

2. It is not logical to require invocation of utilities such as *cd* because they have no value outside the shell environment or cannot be useful in a child process. However, counter-examples always seemed to be available for even the most unusual cases:

```
find . −type d −exec cd {} \; −exec foo {} \;
```

(which invokes ''foo'' on accessible directories)

```
ps ... | sed ... | xargs kill
```

```
find . −exec true \; −a ...
```

(where ''true'' is used for temporary debugging)

3. It is confusing to have a utility such as *kill* that can easily be in the file system in the base standard, but that requires built-in status for the UPE (for the % job control job ID notation). It was decided that it was more appropriate to describe the required functionality (rather than the implementation) to the system implementors and let them decide how to satisfy it.

On the other hand, it was realized that any distinction like this between utilities was not useful to applications, and that the cost to correct it was small. These arguments were ultimately the most effective.

There were varying reasons for including utilities in the table of built-ins:

alias, *fc*, *unalias*
The functionality of these utilities is performed more simply within the shell itself and that is the model most historical implementations have used.

bg, *fg*, *jobs*
All of the job control-related utilities are eligible for built-in status because that is the model most historical implementations have used.

cd, *getopts*, *newgrp*, *read*, *umask*, *wait*
The functionality of these utilities is performed more simply within the context of the current process. An example can be taken from the usage of the *cd* utility. The purpose of the utility is to change the working directory for subsequent operations. The actions of *cd* affect the process in which *cd* is executed and all subsequent child processes of that process. Based on the ISO POSIX-1 standard p1 process model, changes in the process environment of a child process have no effect on the parent process. If the *cd* utility were executed from a child process, the working directory change would be effective only in the child process. Child processes initiated subsequent to the child process that executed the *cd* utility would not have a changed working directory relative to the parent process.

command
This utility was placed in the table primarily to protect scripts that are concerned about their *PATH* being manipulated. The ''secure'' shell script example in the *command* utility in the Shell and Utilities volume of IEEE Std 1003.1-200x would not be possible if a *PATH* change retrieved an alien version of *command*. (An alternative would have been to implement *getconf* as a built-in, but the standard developers considered that it carried too many changing configuration strings to require in the shell.)

kill
Since *kill* provides optional job control functionality using shell notation (`%1`, `%2`, and so on), some implementations would find it extremely difficult to provide this outside the shell.

true, *false*
These are in the table as a courtesy to programmers who wish to use the ''**while** \ *true*'' shell construct without protecting *true* from *PATH* searches. (It is acknowledged that ''**while** \ :'' also works, but the idiom with *true* is historically pervasive.)

All utilities, including those in the table, are accessible via the *system*( ) and *popen*( ) functions in the System Interfaces volume of IEEE Std 1003.1-200x. There are situations where the return functionality of *system*( ) and *popen*( ) is not desirable. Applications that require the exit status of the invoked utility will not be able to use *system*( ) or *popen*( ), since the exit status returned is that of the command language interpreter rather than that of the invoked utility. The alternative for such applications is the use of the *exec* family.

C.2 Shell Command Language

C.2.1 Shell Introduction

The System V shell was selected as the starting point for the Shell and Utilities volume of IEEE Std 1003.1-200x. The BSD C shell was excluded from consideration for the following reasons:

- Most historically portable shell scripts assume the Version 7 Bourne shell, from which the System V shell is derived.
- The majority of tutorial materials on shell programming assume the System V shell.

The construct `"#!"` is reserved for implementations wishing to provide that extension. If it were not reserved, the Shell and Utilities volume of IEEE Std 1003.1-200x would disallow it by forcing it to be a comment. As it stands, a strictly conforming application must not use `"#!"` as the first two characters of the file.

C.2.2 Quoting

There is no additional rationale provided for this section.

C.2.2.1 Escape Character (Backslash)

There is no additional rationale provided for this section.

C.2.2.2 Single-Quotes

A backslash cannot be used to escape a single-quote in a single-quoted string. An embedded quote can be created by writing, for example: `"'a'\''b'"`, which yields `"a'b"`. (See the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.5, Field Splitting for a better understanding of how portions of words are either split into fields or remain concatenated.) A single token can be made up of concatenated partial strings containing all three kinds of quoting or escaping, thus permitting any combination of characters.

C.2.2.3 Double-Quotes

The escaped <newline> used for line continuation is removed entirely from the input and is not replaced by any white space. Therefore, it cannot serve as a token separator.

In double-quoting, if a backslash is immediately followed by a character that would be interpreted as having a special meaning, the backslash is deleted and the subsequent character is taken literally. If a backslash does not precede a character that would have a special meaning, it is left in place unmodified and the character immediately following it is also left unmodified. Thus, for example:

```
"\$"   →   $
"\a"   →   \a
```

It would be desirable to include the statement ‘‘The characters from an enclosed `"${"` to the matching `’}’` shall not be affected by the double quotes’’, similar to the one for `"$()"`. However, historical practice in the System V shell prevents this.

The requirement that double-quotes be matched inside `"${...}"` within double-quotes and the rule for finding the matching `’}’` in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.2, Parameter Expansion eliminate several subtle inconsistencies in expansion for historical shells in rare cases; for example:

```
"${foo-bar"}
```

yields **bar** when **foo** is not defined, and is an invalid substitution when **foo** is defined, in many historical shells. The differences in processing the `"${...}"` form have led to inconsistencies between historical systems. A consequence of this rule is that single-quotes cannot be used to quote the `’}’` within `"${...}"`; for example:

```
unset bar
foo="${bar–’}’}"
```

is invalid because the `"${...}"` substitution contains an unpaired unescaped single-quote. The backslash can be used to escape the `’}’` in this example to achieve the desired result:

```
unset bar
foo="${bar–\}}"
```

The differences in processing the `"${...}"` form have led to inconsistencies between the historical System V shell, BSD, and KornShells, and the text in the Shell and Utilities volume of IEEE Std 1003.1-200x is an attempt to converge them without breaking too many applications. The only alternative to this compromise between shells would be to make the behavior unspecified whenever the literal characters `’’’`, `’{’`, `’}’`, and `’"’` appear within `"${...}"`. To write a portable script that uses these values, a user would have to assign variables; for example:

```
squote=\’ dquote=\" lbrace=’{’ rbrace=’}’
${foo–$squote$rbrace$squote}
```

rather than:

```
${foo–"’}’"}
```

Some implementations have allowed the end of the word to terminate the backquoted command substitution, such as in:

```
"‘echo hello"
```

This usage is undefined; the matching backquote is required by the Shell and Utilities volume of IEEE Std 1003.1-200x. The other undefined usage can be illustrated by the example:

```
sh –c ’‘ echo "foo‘’
```

The description of the recursive actions involving command substitution can be illustrated with an example. Upon recognizing the introduction of command substitution, the shell parses input (in a new context), gathering the source for the command substitution until an unbalanced `’)’` or `’‘’` is located. For example, in the following:

```
echo "$(date; echo "
    one" )"
```

the double-quote following the *echo* does not terminate the first double-quote; it is part of the command substitution script. Similarly, in:

```
echo "$(echo *)"
```

the asterisk is not quoted since it is inside command substitution; however:

```
echo "$(echo "*")"
```

is quoted (and represents the asterisk character itself).

C.2.3 Token Recognition

The `"(("` and `"))"` symbols are control operators in the KornShell, used for an alternative syntax of an arithmetic expression command. A conforming application cannot use `"(("` as a single token (with the exception of the `"$(("` form for shell arithmetic).

On some implementations, the symbol `"(("` is a control operator; its use produces unspecified results. Applications that wish to have nested subshells, such as:

```
((echo Hello);(echo World))
```

shall separate the `"(("` characters into two tokens by including white space between them. Some systems may treat these as invalid arithmetic expressions instead of subshells.

Certain combinations of characters are invalid in portable scripts, as shown in the grammar. Implementations may use these combinations (such as `"|&"`) as valid control operators. Portable scripts cannot rely on receiving errors in all cases where this volume of IEEE Std 1003.1-200x indicates that a syntax is invalid.

The (3) rule about combining characters to form operators is not meant to preclude systems from extending the shell language when characters are combined in otherwise invalid ways. Conforming applications cannot use invalid combinations, and test suites should not penalize systems that take advantage of this fact. For example, the unquoted combination `"|&"` is not valid in a POSIX script, but has a specific KornShell meaning.

The (10) rule about `'#'` as the current character is the first in the sequence in which a new token is being assembled. The `'#'` starts a comment only when it is at the beginning of a token. This rule is also written to indicate that the search for the end-of-comment does not consider escaped <newline> specially, so that a comment cannot be continued to the next line.

C.2.3.1 Alias Substitution

The alias capability was added in the UPE because it is widely used in historical implementations by interactive users.

The definition of *alias name* precludes an alias name containing a slash character. Since the text applies to the command words of simple commands, reserved words (in their proper places) cannot be confused with aliases.

The placement of alias substitution in token recognition makes it clear that it precedes all of the word expansion steps.

An example concerning trailing <blank>s and reserved words follows. If the user types:

```
$ alias foo="/bin/ls "
$ alias while="/"
```

The effect of executing:

```
$ while true
> do
> echo "Hello, World"
> done
```

is a never-ending sequence of `"Hello, World"` strings to the screen. However, if the user types:

```
$ foo while
```

the result is an *ls* listing of **/**. Since the alias substitution for **foo** ends in a <space>, the next word is checked for alias substitution. The next word, **while**, has also been aliased, so it is substituted as well. Since it is not in the proper position as a command word, it is not recognized as a reserved word.

If the user types:

```
$ foo; while
```

while retains its normal reserved-word properties.

C.2.4 Reserved Words

All reserved words are recognized syntactically as such in the contexts described. However, note that **in** is the only meaningful reserved word after a **case** or **for**; similarly, **in** is not meaningful as the first word of a simple command.

Reserved words are recognized only when they are delimited (that is, meet the definition of the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.435, Word), whereas operators are themselves delimiters. For instance, `'('` and `')'` are control operators, so that no <space> is needed in (*list*). However, `'{'` and `'}'` are reserved words in { *list*;}, so that in this case the leading <space> and semicolon are required.

The list of unspecified reserved words is from the KornShell, so conforming applications cannot use them in places a reserved word would be recognized. This list contained **time** in early proposals, but it was removed when the *time* utility was selected for the Shell and Utilities volume of IEEE Std 1003.1-200x.

There was a strong argument for promoting braces to operators (instead of reserved words), so they would be syntactically equivalent to subshell operators. Concerns about compatibility outweighed the advantages of this approach. Nevertheless, conforming applications should consider quoting `'{'` and `'}'` when they represent themselves.

The restriction on ending a name with a colon is to allow future implementations that support named labels for flow control; see the RATIONALE for the *break* built-in utility .

It is possible that a future version of the Shell and Utilities volume of IEEE Std 1003.1-200x may require that `'{'` and `'}'` be treated individually as control operators, although the token `"{}"` will probably be a special-case exemption from this because of the often-used *find*{} construct.

C.2.5 Parameters and Variables

C.2.5.1 Positional Parameters

There is no additional rationale provided for this section.

C.2.5.2 Special Parameters

Most historical implementations implement subshells by forking; thus, the special parameter `'$'` does not necessarily represent the process ID of the shell process executing the commands since the subshell execution environment preserves the value of `'$'`.

If a subshell were to execute a background command, the value of `"$!"` for the parent would not change. For example:

```
(
date &
echo $!
)
echo $!
```

would echo two different values for `"$!"`.

The `"$–"` special parameter can be used to save and restore *set* options:

```
Save=$(echo $– | sed 's/[ics]//g')
...
set +aCefnuvx
if [ –n "$Save" ]; then
    set –$Save
fi
```

The three options are removed using *sed* in the example because they may appear in the value of `"$–"` (from the *sh* command line), but are not valid options to *set*.

The descriptions of parameters `'*'` and `'@'` assume the reader is familiar with the field splitting discussion in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.5, Field Splitting and understands that portions of the word remain concatenated unless there is some reason to split them into separate fields.

Some examples of the `'*'` and `'@'` properties, including the concatenation aspects:

```
set "abc" "def ghi" "jkl"

echo $*     => "abc" "def" "ghi" "jkl"
echo "$*"   => "abc def ghi jkl"
echo $@     => "abc" "def" "ghi" "jkl"
```

but:

```
echo "$@"       => "abc" "def ghi" "jkl"
echo "xx$@yy"   => "xxabc" "def ghi" "jklyy"
echo "$@$@"     => "abc" "def ghi" "jklabc" "def ghi" "jkl"
```

In the preceding examples, the double-quote characters that appear after the `"=>"` do not appear in the output and are used only to illustrate word boundaries.

The following example illustrates the effect of setting *IFS* to a null string:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

C.2.5.3 *Shell Variables*

See the discussion of *IFS* in Section C.2.6.5 (on page 3529).

The prohibition on *LC_CTYPE* changes affecting lexical processing protects the shell implementor (and the shell programmer) from the ill effects of changing the definition of <blank> or the set of alphabetic characters in the current environment. It would probably not be feasible to write a compiled version of a shell script without this rule. The rule applies only to the current invocation of the shell and its subshells—invoking a shell script or performing *exec sh* would subject the new shell to the changes in *LC_CTYPE*.

Other common environment variables used by historical shells are not specified by the Shell and Utilities volume of IEEE Std 1003.1-200x, but they should be reserved for the historical uses.

Tilde expansion for components of the *PATH* in an assignment such as:

```
PATH=˜hlj/bin:˜dwc/bin:$PATH
```

is a feature of some historical shells and is allowed by the wording of the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.6.1, Tilde Expansion. Note that the tildes are expanded during the assignment to *PATH*, not when *PATH* is accessed during command search.

The following entries represent additional information about variables included in the Shell and Utilities volume of IEEE Std 1003.1-200x, or rationale for common variables in use by shells that have been excluded:

_	(Underscore.) While underscore is historical practice, its overloaded usage in the KornShell is confusing, and it has been omitted from the Shell and Utilities volume of IEEE Std 1003.1-200x.
ENV	This variable can be used to set aliases and other items local to the invocation of a shell. The file referred to by *ENV* differs from **$HOME/.profile** in that **.profile** is typically executed at session start-up, whereas the *ENV* file is executed at the beginning of each shell invocation. The *ENV* value is interpreted in a manner similar to a dot script, in that the commands are executed in the current environment and the file needs to be readable, but not executable. However, unlike dot scripts, no *PATH* searching is performed. This is used as a guard against Trojan Horse security breaches.
ERRNO	This variable was omitted from the Shell and Utilities volume of IEEE Std 1003.1-200x because the values of error numbers are not defined in IEEE Std 1003.1-200x in a portable manner.
FCEDIT	Since this variable affects only the *fc* utility, it has been omitted from this more global place. The value of *FCEDIT* does not affect the command line editing mode in the shell; see the description of *set* −**o** *vi* in the *set* built-in utility.

PS1 This variable is used for interactive prompts. Historically, the ‘‘superuser’’ has had a prompt of `'#'`. Since privileges are not required to be monolithic, it is difficult to define which privileges should cause the alternate prompt. However, a sufficiently powerful user should be reminded of that power by having an alternate prompt.

PS3 This variable is used by the KornShell for the *select* command. Since the POSIX shell does not include *select*, *PS3* was omitted.

PS4 This variable is used for shell debugging. For example, the following script:

```
PS4='[${LINENO}]+ '
set −x
echo Hello
```

writes the following to standard error:

```
[3]+ echo Hello
```

RANDOM This pseudo-random number generator was not seen as being useful to interactive users.

SECONDS Although this variable is sometimes used with *PS1* to allow the display of the current time in the prompt of the user, it is not one that would be manipulated frequently enough by an interactive user to include in the Shell and Utilities volume of IEEE Std 1003.1-200x.

C.2.6 Word Expansions

Step (2) refers to the ‘‘portions of fields generated by step (1)’’. For example, if the word being expanded were `"$x+$y"` and *IFS*=+, the word would be split only if `"$x"` or `"$y"` contained `'+'`; the `'+'` in the original word was not generated by step (1).

IFS is used for performing field splitting on the results of parameter and command substitution; it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields during field splitting, but this has severe problems because the shell can no longer parse its own script. There are also important security implications caused by this behavior. All useful applications of *IFS* use it for parsing input of the *read* utility and for splitting the results of parameter and command substitution.

The rule concerning expansion to a single field requires that if **foo**=**abc** and **bar**=**def**, that:

```
"$foo""$bar"
```

expands to the single field:

```
abcdef
```

The rule concerning empty fields can be illustrated by:

```
$    unset foo
$    set $foo bar ’’ xyz "$foo" abc
$    for i
>    do
>        echo "-$i-"
>    done
-bar-
--
-xyz-
--
-abc-
```

Step (1) indicates that parameter expansion, command substitution, and arithmetic expansion are all processed simultaneously as they are scanned. For example, the following is valid arithmetic:

```
x=1
echo $(( $(echo 3)+$x ))
```

An early proposal stated that tilde expansion preceded the other steps, but this is not the case in known historical implementations; if it were, and if a referenced home directory contained a '$' character, expansions would result within the directory name.

C.2.6.1 Tilde Expansion

Tilde expansion generally occurs only at the beginning of words, but an exception based on historical practice has been included:

```
PATH=/posix/bin:˜dgk/bin
```

This is eligible for tilde expansion because tilde follows a colon and none of the relevant characters is quoted. Consideration was given to prohibiting this behavior because any of the following are reasonable substitutes:

```
PATH=$(printf %s ˜karels/bin : ˜bostic/bin)
```

```
for Dir in ˜maart/bin ˜srb/bin ...
do
    PATH=${PATH:+$PATH:}$Dir
done
```

In the first command, explicit colons are used for each directory. In all cases, the shell performs tilde expansion on each directory because all are separate words to the shell.

Note that expressions in operands such as:

```
make -k mumble LIBDIR=˜chet/lib
```

do not qualify as shell variable assignments, and tilde expansion is not performed (unless the command does so itself, which *make* does not).

Because of the requirement that the word is not quoted, the following are not equivalent; only the last causes tilde expansion:

```
\˜hlj/   ~h\lj/   ~"hlj"/   ~hlj\/   ~hlj/
```

In an early proposal, tilde expansion occurred following any unquoted equals sign or colon, but this was removed because of its complexity and to avoid breaking commands such as:

```
rcp hostname:˜marc/.profile .
```

A suggestion was made that the special sequence "$˜" should be allowed to force tilde expansion anywhere. Since this is not historical practice, it has been left for future implementations to evaluate. (The description in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.2, Quoting requires that a dollar sign be quoted to represent itself, so the "$˜" combination is already unspecified.)

The results of giving tilde with an unknown login name are undefined because the KornShell "˜+" and "˜−" constructs make use of this condition, but in general it is an error to give an incorrect login name with tilde. The results of having *HOME* unset are unspecified because some historical shells treat this as an error.

C.2.6.2 *Parameter Expansion*

The rule for finding the closing '}' in "${...}" is the one used in the KornShell and is upwardly-compatible with the Bourne shell, which does not determine the closing '}' until the word is expanded. The advantage of this is that incomplete expansions, such as:

```
${foo
```

can be determined during tokenization, rather than during expansion.

The string length and substring capabilities were included because of the demonstrated need for them, based on their usage in other shells, such as C shell and KornShell.

Historical versions of the KornShell have not performed tilde expansion on the word part of parameter expansion; however, it is more consistent to do so.

C.2.6.3 *Command Substitution*

The "$()" form of command substitution solves a problem of inconsistent behavior when using backquotes. For example:

Command	Output
`echo '\$x'`	`\$x`
``echo `echo '\$x'` ``	`$x`
`echo $(echo '\$x')`	`\$x`

Additionally, the backquoted syntax has historical restrictions on the contents of the embedded command. While the newer "$()" form can process any kind of valid embedded script, the backquoted form cannot handle some valid scripts that include backquotes. For example, these otherwise valid embedded scripts do not work in the left column, but do work on the right:

```
echo `                         echo $(
cat <<\eof                     cat <<\eof
a here-doc with `              a here-doc with )
eof                            eof
`                              )

echo `                         echo $(
echo abc # a comment with `    echo abc # a comment with )
`                              )

echo `                         echo $(
echo '`'                       echo ')'
`                              )
```

Because of these inconsistent behaviors, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts.

The KornShell feature:

> If *command* is of the form <*word*, *word* is expanded to generate a pathname, and the value of the command substitution is the contents of this file with any trailing <newline>s deleted.

was omitted from the Shell and Utilities volume of IEEE Std 1003.1-200x because $(*cat word*) is an appropriate substitute. However, to prevent breaking numerous scripts relying on this feature, it is unspecified to have a script within `"$()"` that has only redirections.

The requirement to separate `"$("` and `'('` when a single subshell is command-substituted is to avoid any ambiguities with arithmetic expansion.

C.2.6.4 *Arithmetic Expansion*

The `"(())"` form of KornShell arithmetic in early proposals was omitted. The standard developers concluded that there was a strong desire for some kind of arithmetic evaluator to replace *expr*, and that relating it to `'$'` makes it work well with the standard shell language, and it provides access to arithmetic evaluation in places where accessing a utility would be inconvenient.

The syntax and semantics for arithmetic were changed for the ISO/IEC 9945-2:1993 standard. The language is essentially a pure arithmetic evaluator of constants and operators (excluding assignment) and represents a simple subset of the previous arithmetic language (which was derived from the KornShell `"(())"` construct). The syntax was changed from that of a command denoted by ((*expression*)) to an expansion denoted by $((*expression*)). The new form is a dollar expansion (`'$'`) that evaluates the expression and substitutes the resulting value. Objections to the previous style of arithmetic included that it was too complicated, did not fit in well with the use of variables in the shell, and its syntax conflicted with subshells. The justification for the new syntax is that the shell is traditionally a macro language, and if a new feature is to be added, it should be accomplished by extending the capabilities presented by the current model of the shell, rather than by inventing a new one outside the model; adding a new dollar expansion was perceived to be the most intuitive and least destructive way to add such a new capability.

In early proposals, a form $[*expression*] was used. It was functionally equivalent to the `"$(())"` of the current text, but objections were lodged that the 1988 KornShell had already implemented `"$(())"` and there was no compelling reason to invent yet another syntax. Furthermore, the `"$[]"` syntax had a minor incompatibility involving the patterns in **case** statements.

The portion of the ISO C standard arithmetic operations selected corresponds to the operations historically supported in the KornShell.

It was concluded that the *test* command (**[**) was sufficient for the majority of relational arithmetic tests, and that tests involving complicated relational expressions within the shell are rare, yet could still be accommodated by testing the value of `"$(())"` itself. For example:

```
# a complicated relational expression
while [ $(( (($x + $y)/($a * $b)) < ($foo*$bar) )) −ne 0 ]
```

or better yet, the rare script that has many complex relational expressions could define a function like this:

```
val() {
    return $((!$1))
}
```

and complicated tests would be less intimidating:

```
while val $(( (($x + $y)/($a * $b)) < ($foo*$bar) ))
do
    # some calculations
done
```

A suggestion that was not adopted was to modify *true* and *false* to take an optional argument, and *true* would exit true only if the argument was non-zero, and *false* would exit false only if the argument was non-zero:

```
while true $(($x > 5 && $y <= 25))
```

There is a minor portability concern with the new syntax. The example $((2+2)) could have been intended to mean a command substitution of a utility named 2+2 in a subshell. The standard developers considered this to be obscure and isolated to some KornShell scripts (because `"$()"` command substitution existed previously only in the KornShell). The text on command substitution requires that the `"$("` and `'('` be separate tokens if this usage is needed.

An example such as:

```
echo $((echo hi);(echo there))
```

should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. However, as indicated by the Base Definitions volume of IEEE Std 1003.1-200x, Section 3.112, Control Operator, a conforming application must separate two adjacent parentheses with white space to indicate nested subshells.

Although the ISO/IEC 9899:1999 standard now requires support for **long long** and allows extended integer types with higher ranks, IEEE Std 1003.1-200x only requires arithmetic expansions to support **signed long** integer arithmetic. Implementations are encouraged to support signed integer values at least as large as the size of the largest file allowed on the implementation.

Implementations are also allowed to perform floating-point evaluations as long as an application won't see different results for expressions that would not overflow **signed long** integer expression evaluation. (This includes appropriate truncation of results to integer values.)

Changes made in response to IEEE PASC Interpretation 1003.2 #208 removed the requirement that the integer constant suffixes `l` and `L` had to be recognized. The ISO POSIX-2:1993 standard didn't require the `u`, `ul`, `uL`, `U`, `Ul`, `UL`, `lu`, `lU`, `Lu`, and `LU` suffixes since only signed integer arithmetic was required. Since all arithmetic expressions were treated as handling **signed long** integer types anyway, the `l` and `L` suffixes were redundant. No known scripts used them and some historic shells didn't support them. When the ISO/IEC 9899:1999 standard was used as the basis for the description of arithmetic processing, the `ll` and `LL` suffixes and combinations were also not required. Implementations are still free to accept any or all of these suffices, but are not required to do so.

There was also some confusion as to whether the shell was required to recognize character constants. Syntactically, character constants were required to be recognized, but the requirements for the handling of backslash (`"\\"`) and quote (`'\''`) characters (needed to specify character constants) within an arithmetic expansion were ambiguous. Furthermore, no known shells supported them. Changes made in response to IEEE PASC Interpretation 1003.2

#208 removed the requirement to support them (if they were indeed required before). IEEE Std 1003.1-200x clearly does not require support for character constants.

C.2.6.5 *Field Splitting*

The operation of field splitting using *IFS*, as described in early proposals, was based on the way the KornShell splits words, but it is incompatible with other common versions of the shell. However, each has merit, and so a decision was made to allow both. If the *IFS* variable is unset or is <space><tab><newline>, the operation is equivalent to the way the System V shell splits words. Using characters outside the <space><tab><newline> set yields the KornShell behavior, where each of the non-<space><tab><newline>s is significant. This behavior, which affords the most flexibility, was taken from the way the original *awk* handled field splitting.

Rule (3) can be summarized as a pseudo-ERE:

```
(s*ns*|s+)
```

where *s* is an *IFS* white space character and *n* is a character in the *IFS* that is not white space. Any string matching that ERE delimits a field, except that the *s*+ form does not delimit fields at the beginning or the end of a line. For example, if *IFS* is <space>/<comma>/<tab>, the string:

```
<space><space>red<space><space>,<space>white<space>blue
```

yields the three colors as the delimited fields.

C.2.6.6 *Pathname Expansion*

There is no additional rationale provided for this section.

C.2.6.7 *Quote Removal*

There is no additional rationale provided for this section.

C.2.7 Redirection

In the System Interfaces volume of IEEE Std 1003.1-200x, file descriptors are integers in the range 0–({OPEN_MAX}–1). The file descriptors discussed in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.7, Redirection are that same set of small integers.

Having multi-digit file descriptor numbers for I/O redirection can cause some obscure compatibility problems. Specifically, scripts that depend on an example command:

```
echo 22>/dev/null
```

echoing 2 to standard error or 22 to standard output are no longer portable. However, the file descriptor number still must be delimited from the preceding text. For example:

```
cat file2>foo
```

writes the contents of **file2**, not the contents of **file**.

The `">|"` format of output redirection was adopted from the KornShell. Along with the *noclobber* option, *set* −**C**, it provides a safety feature to prevent inadvertent overwriting of existing files. (See the RATIONALE for the *pathchk* utility for why this step was taken.) The restriction on regular files is historical practice.

The System V shell and the KornShell have differed historically on pathname expansion of *word*; the former never performed it, the latter only when the result was a single field (file). As a compromise, it was decided that the KornShell functionality was useful, but only as a shorthand device for interactive users. No reasonable shell script would be written with a command such as:

```
cat foo > a*
```

Thus, shell scripts are prohibited from doing it, while interactive users can select the shell with which they are most comfortable.

The construct 2>&1 is often used to redirect standard error to the same file as standard output. Since the redirections take place beginning to end, the order of redirections is significant. For example:

```
ls > foo 2>&1
```

directs both standard output and standard error to file **foo**. However:

```
ls 2>&1 > foo
```

only directs standard output to file **foo** because standard error was duplicated as standard output before standard output was directed to file **foo**.

The "<>" operator could be useful in writing an application that worked with several terminals, and occasionally wanted to start up a shell. That shell would in turn be unable to run applications that run from an ordinary controlling terminal unless it could make use of "<>" redirection. The specific example is a historical version of the pager *more*, which reads from standard error to get its commands, so standard input and standard output are both available for their usual usage. There is no way of saying the following in the shell without "<>":

```
cat food | more – >/dev/tty03 2<>/dev/tty03
```

Another example of "<>" is one that opens **/dev/tty** on file descriptor 3 for reading and writing:

```
exec 3<> /dev/tty
```

An example of creating a lock file for a critical code region:

```
set –C
until    2> /dev/null > lockfile
do       sleep 30
done
set +C
perform critical function
rm lockfile
```

Since **/dev/null** is not a regular file, no error is generated by redirecting to it in *noclobber* mode.

Tilde expansion is not performed on a here-document because the data is treated as if it were enclosed in double quotes.

C.2.7.1 Redirecting Input

There is no additional rationale provided for this section.

C.2.7.2 Redirecting Output

There is no additional rationale provided for this section.

C.2.7.3 Appending Redirected Output

Note that when a file is opened (even with the O_APPEND flag set), the initial file offset for that file is set to the beginning of the file. Some historic shells set the file offset to the current end-of-file when append mode shell redirection was used, but this is not allowed by IEEE Std 1003.1-200x.

C.2.7.4 Here-Document

There is no additional rationale provided for this section.

C.2.7.5 Duplicating an Input File Descriptor

There is no additional rationale provided for this section.

C.2.7.6 Duplicating an Output File Descriptor

There is no additional rationale provided for this section.

C.2.7.7 Open File Descriptors for Reading and Writing

There is no additional rationale provided for this section.

C.2.8 Exit Status and Errors

C.2.8.1 Consequences of Shell Errors

There is no additional rationale provided for this section.

C.2.8.2 Exit Status for Commands

There is a historical difference in *sh* and *ksh* non-interactive error behavior. When a command named in a script is not found, some implementations of *sh* exit immediately, but *ksh* continues with the next command. Thus, the Shell and Utilities volume of IEEE Std 1003.1-200x says that the shell ‘‘may’’ exit in this case. This puts a small burden on the programmer, who has to test for successful completion following a command if it is important that the next command not be executed if the previous command was not found. If it is important for the command to have been found, it was probably also important for it to complete successfully. The test for successful completion would not need to change.

Historically, shells have returned an exit status of $128+n$, where n represents the signal number. Since signal numbers are not standardized, there is no portable way to determine which signal caused the termination. Also, it is possible for a command to exit with a status in the same range of numbers that the shell would use to report that the command was terminated by a signal. Implementations are encouraged to choose exit values greater than 256 to indicate programs that terminate by a signal so that the exit status cannot be confused with an exit status generated by a normal termination.

Historical shells make the distinction between ‘‘utility not found’’ and ‘‘utility found but cannot execute’’ in their error messages. By specifying two seldomly used exit status values for these cases, 127 and 126 respectively, this gives an application the opportunity to make use of this distinction without having to parse an error message that would probably change from locale to locale. The *command*, *env*, *nohup*, and *xargs* utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x have also been specified to use this convention.

When a command fails during word expansion or redirection, most historical implementations exit with a status of 1. However, there was some sentiment that this value should probably be much higher so that an application could distinguish this case from the more normal exit status values. Thus, the language ‘‘greater than zero’’ was selected to allow either method to be implemented.

C.2.9 Shell Commands

A description of an ‘‘empty command’’ was removed from an early proposal because it is only relevant in the cases of *sh* −**c** `""`, *system*(`""`), or an empty shell-script file (such as the implementation of *true* on some historical systems). Since it is no longer mentioned in the Shell and Utilities volume of IEEE Std 1003.1-200x, it falls into the silently unspecified category of behavior where implementations can continue to operate as they have historically, but conforming applications do not construct empty commands. (However, note that *sh* does explicitly state an exit status for an empty string or file.) In an interactive session or a script with other commands, extra <newline>s or semicolons, such as;

```
$ false
$
$ echo $?
1
```

would not qualify as the empty command described here because they would be consumed by other parts of the grammar.

C.2.9.1 Simple Commands

The enumerated list is used only when the command is actually going to be executed. For example, in:

```
true || $foo *
```

no expansions are performed.

The following example illustrates both how a variable assignment without a command name affects the current execution environment, and how an assignment with a command name only affects the execution environment of the command:

```
$ x=red
$ echo $x
red
$ export x
$ sh −c ’echo $x’
red
$ x=blue sh −c ’echo $x’
blue
$ echo $x
red
```

This next example illustrates that redirections without a command name are still performed:

```
$ ls foo
ls: foo: no such file or directory
$ > foo
$ ls foo
foo
```

A command without a command name, but one that includes a command substitution, has an exit status of the last command substitution that the shell performed. For example:

```
if      x=$(command)
then    ...
fi
```

An example of redirections without a command name being performed in a subshell shows that the here-document does not disrupt the standard input of the **while** loop:

```
IFS=:
while    read a b
do       echo $a
         <<−eof
         Hello
         eof
done </etc/passwd
```

Some examples of commands without command names in AND-OR lists:

```
> foo || {
    echo "error: foo cannot be created" >&2
    exit 1
}

# set saved if /vmunix.save exists
test −f /vmunix.save && saved=1
```

Command substitution and redirections without command names both occur in subshells, but they are not necessarily the same ones. For example, in:

```
exec 3> file
var=$(echo foo >&3) 3>&1
```

it is unspecified whether **foo** is echoed to the file or to standard output.

Command Search and Execution

This description requires that the shell can execute shell scripts directly, even if the underlying system does not support the common `"#!"` interpreter convention. That is, if file **foo** contains shell commands and is executable, the following executes **foo**:

```
./foo
```

The command search shown here does not match all historical implementations. A more typical sequence has been:

- Any built-in (special or regular)
- Functions
- Path search for executable files

But there are problems with this sequence. Since the programmer has no idea in advance which utilities might have been built into the shell, a function cannot be used to override portably a utility of the same name. (For example, a function named *cd* cannot be written for many historical systems.) Furthermore, the *PATH* variable is partially ineffective in this case, and only a pathname with a slash can be used to ensure a specific executable file is invoked.

After the *execve*( ) failure described, the shell normally executes the file as a shell script. Some implementations, however, attempt to detect whether the file is actually a script and not an executable from some other architecture. The method used by the KornShell is allowed by the text that indicates non-text files may be bypassed.

The sequence selected for the Shell and Utilities volume of IEEE Std 1003.1-200x acknowledges that special built-ins cannot be overridden, but gives the programmer full control over which versions of other utilities are executed. It provides a means of suppressing function lookup (via the *command* utility) for the user's own functions and ensures that any regular built-ins or functions provided by the implementation are under the control of the path search. The mechanisms for associating built-ins or functions with executable files in the path are not specified by the Shell and Utilities volume of IEEE Std 1003.1-200x, but the wording requires that if either is implemented, the application is not able to distinguish a function or built-in from an executable (other than in terms of performance, presumably). The implementation ensures that all effects specified by the Shell and Utilities volume of IEEE Std 1003.1-200x resulting from the invocation of the regular built-in or function (interaction with the environment, variables, traps, and so on) are identical to those resulting from the invocation of an executable file.

Examples

Consider three versions of the *ls* utility:

1. The application includes a shell function named *ls*.
2. The user writes a utility named *ls* and puts it in **/fred/bin**.
3. The example implementation provides *ls* as a regular shell built-in that is invoked (either by the shell or directly by *exec*) when the path search reaches the directory **/posix/bin**.

If *PATH*=**/posix/bin**, various invocations yield different versions of *ls*:

Invocation	**Version of *ls***
ls (from within application script)	(1) function
command ls (from within application script)	(3) built-in
ls (from within makefile called by application)	(3) built-in
system("*ls*")	(3) built-in
PATH="**/fred/bin**:$*PATH*" *ls*	(2) user's version

C.2.9.2 Pipelines

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
$ command1 2>&1 | command2
```

sends both the standard output and standard error of *command1* to the standard input of *command2*.

The reserved word **!** allows more flexible testing using AND and OR lists.

It was suggested that it would be better to return a non-zero value if any command in the pipeline terminates with non-zero status (perhaps the bitwise-inclusive OR of all return values). However, the choice of the last-specified command semantics are historical practice and would cause applications to break if changed. An example of historical behavior:

```
$ sleep 5 | (exit 4)
$ echo $?
4
$ (exit 4) | sleep 5
$ echo $?
0
```

C.2.9.3 Lists

The equal precedence of "&&" and "||" is historical practice. The standard developers evaluated the model used more frequently in high-level programming languages, such as C, to allow the shell logical operators to be used for complex expressions in an unambiguous way, but they could not allow historical scripts to break in the subtle way unequal precedence might cause. Some arguments were posed concerning the "{}" or "()" groupings that are required historically. There are some disadvantages to these groupings:

- The "()" can be expensive, as they spawn other processes on some implementations. This performance concern is primarily an implementation issue.
- The "{}" braces are not operators (they are reserved words) and require a trailing space after each '{', and a semicolon before each '}'. Most programmers (and certainly interactive users) have avoided braces as grouping constructs because of the problematic syntax required. Braces were not changed to operators because that would generate compatibility issues even greater than the precedence question; braces appear outside the context of a keyword in many shell scripts.

IEEE PASC Interpretation 1003.2 #204 is applied, clarifying that the operators "&&" and "||" are evaluated with left associativity.

Asynchronous Lists

The grammar treats a construct such as:

```
foo & bar & bam &
```

as one ‘‘asynchronous list’’, but since the status of each element is tracked by the shell, the term ‘‘element of an asynchronous list’’ was introduced to identify just one of the **foo**, **bar**, or **bam** portions of the overall list.

Unless the implementation has an internal limit, such as {CHILD_MAX}, on the retained process IDs, it would require unbounded memory for the following example:

```
while true
do      foo & echo $!
done
```

The treatment of the signals SIGINT and SIGQUIT with asynchronous lists is described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.11, Signals and Error Handling.

Since the connection of the input to the equivalent of **/dev/null** is considered to occur before redirections, the following script would produce no output:

```
exec < /etc/passwd
cat <&0 &
wait
```

Sequential Lists

There is no additional rationale provided for this section.

AND Lists

There is no additional rationale provided for this section.

OR Lists

There is no additional rationale provided for this section.

C.2.9.4 *Compound Commands*

Grouping Commands

The semicolon shown in {*compound-list*;} is an example of a control operator delimiting the **}** reserved word. Other delimiters are possible, as shown in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.10, Shell Grammar; <newline> is frequently used.

A proposal was made to use the **<do-done>** construct in all cases where command grouping in the current process environment is performed, identifying it as a construct for the grouping commands, as well as for shell functions. This was not included because the shell already has a grouping construct for this purpose (`"{}"`), and changing it would have been counter-productive.

For Loop

The format is shown with generous usage of <newline>s. See the grammar in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.10, Shell Grammar for a precise description of where <newline>s and semicolons can be interchanged.

Some historical implementations support `'{'` and `'}'` as substitutes for **do** and **done**. The standard developers chose to omit them, even as an obsolescent feature. (Note that these substitutes were only for the **for** command; the **while** and **until** commands could not use them historically because they are followed by compound-lists that may contain `"{...}"` grouping commands themselves.)

The reserved word pair **do** ... **done** was selected rather than **do** ... **od** (which would have matched the spirit of **if** ... **fi** and **case** ... **esac**) because *od* is already the name of a standard utility.

PASC Interpretation 1003.2 #169 has been applied changing the grammar.

Case Conditional Construct

An optional left parenthesis before *pattern* was added to allow numerous historical KornShell scripts to conform. At one time, using the leading parenthesis was required if the **case** statement was to be embedded within a `"$()"` command substitution; this is no longer the case with the POSIX shell. Nevertheless, many historical scripts use the left parenthesis, if only because it makes matching-parenthesis searching easier in *vi* and other editors. This is a relatively simple implementation change that is upward-compatible for all scripts.

Consideration was given to requiring *break* inside the *compound-list* to prevent falling through to the next pattern action list. This was rejected as being nonexisting practice. An interesting undocumented feature of the KornShell is that using `";&"` instead of `";;"` as a terminator causes the exact opposite behavior—the flow of control continues with the next *compound-list*.

The pattern `'*'`, given as the last pattern in a **case** construct, is equivalent to the default case in a C-language **switch** statement.

The grammar shows that reserved words can be used as patterns, even if one is the first word on a line. Obviously, the reserved word **esac** cannot be used in this manner.

If Conditional Construct

The precise format for the command syntax is described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.10, Shell Grammar.

While Loop

The precise format for the command syntax is described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.10, Shell Grammar.

Until Loop

The precise format for the command syntax is described in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.10, Shell Grammar.

C.2.9.5 Function Definition Command

The description of functions in an early proposal was based on the notion that functions should behave like miniature shell scripts; that is, except for sharing variables, most elements of an execution environment should behave as if they were a new execution environment, and changes to these should be local to the function. For example, traps and options should be reset on entry to the function, and any changes to them do not affect the traps or options of the caller. There were numerous objections to this basic idea, and the opponents asserted that functions were intended to be a convenient mechanism for grouping common commands that were to be executed in the current execution environment, similar to the execution of the *dot* special built-in.

It was also pointed out that the functions described in that early proposal did not provide a local scope for everything a new shell script would, such as the current working directory, or *umask*, but instead provided a local scope for only a few select properties. The basic argument was that if a local scope is needed for the execution environment, the mechanism already existed: the application can put the commands in a new shell script and call that script. All historical shells that implemented functions, other than the KornShell, have implemented functions that operate in the current execution environment. Because of this, traps and options have a global scope within a shell script. Local variables within a function were considered and included in another early proposal (controlled by the special built-in *local*), but were removed because they do not fit the simple model developed for functions and because there was some opposition to adding yet another new special built-in that was not part of historical practice. Implementations should reserve the identifier *local* (as well as *typeset*, as used in the KornShell) in case this local variable mechanism is adopted in a future version of IEEE Std 1003.1-200x.

A separate issue from the execution environment of a function is the availability of that function to child shells. A few objectors maintained that just as a variable can be shared with child shells by exporting it, so should a function. In early proposals, the *export* command therefore had a **–f** flag for exporting functions. Functions that were exported were to be put into the environment as *name*( )=*value* pairs, and upon invocation, the shell would scan the environment for these and automatically define these functions. This facility was strongly opposed and was omitted. Some of the arguments against exportable functions were as follows:

- There was little historical practice. The Ninth Edition shell provided them, but there was controversy over how well it worked.

- There are numerous security problems associated with functions appearing in the environment of a user and overriding standard utilities or the utilities owned by the application.
- There was controversy over requiring *make* to import functions, where it has historically used an *exec* function for many of its command line executions.
- Functions can be big and the environment is of a limited size. (The counter-argument was that functions are no different from variables in terms of size: there can be big ones, and there can be small ones—and just as one does not export huge variables, one does not export huge functions. However, this might not apply to the average shell-function writer, who typically writes much larger functions than variables.)

As far as can be determined, the functions in the Shell and Utilities volume of IEEE Std 1003.1-200x match those in System V. Earlier versions of the KornShell had two methods of defining functions:

```
function fname { compound-list }
```

and:

```
fname() { compound-list }
```

The latter used the same definition as the Shell and Utilities volume of IEEE Std 1003.1-200x, but differed in semantics, as described previously. The current edition of the KornShell aligns the latter syntax with the Shell and Utilities volume of IEEE Std 1003.1-200x and keeps the former as is.

The name space for functions is limited to that of a *name* because of historical practice. Complications in defining the syntactic rules for the function definition command and in dealing with known extensions such as the `"@()"` usage in the KornShell prevented the name space from being widened to a *word*. Using functions to support synonyms such as the `"!!"` and `'%'` usage in the C shell is thus disallowed to conforming applications, but acceptable as an extension. For interactive users, the aliasing facilities in the Shell and Utilities volume of IEEE Std 1003.1-200x should be adequate for this purpose. It is recognized that the name space for utilities in the file system is wider than that currently supported for functions, if the portable filename character set guidelines are ignored, but it did not seem useful to mandate extensions in systems for so little benefit to conforming applications.

The `"()"` in the function definition command consists of two operators. Therefore, intermixing <blank>s with the *fname*, `'('`, and `')'` is allowed, but unnecessary.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls -l
#
[ "$i" = yes ] && foo() {
    ls -l
}
```

C.2.10 Shell Grammar

There are several subtle aspects of this grammar where conventional usage implies rules about the grammar that in fact are not true.

For *compound_list*, only the forms that end in a *separator* allow a reserved word to be recognized, so usually only a *separator* can be used where a compound list precedes a reserved word (such as **Then**, **Else**, **Do** and **Rbrace**). Explicitly requiring a separator would disallow such valid (if rare) statements as:

```
if (false) then (echo x) else (echo y) fi
```

See the Note under special grammar rule 1.

Concerning the third sentence of rule (1) (''Also, if the parser ...''):

- This sentence applies rather narrowly: when a compound list is terminated by some clear delimiter (such as the closing **fi** of an inner **if_clause**) then it would apply; where the compound list might continue (as in after a `';'`), rule (7a) (and consequently the first sentence of rule (1)) would apply. In many instances the two conditions are identical, but this part of rule (1) does not give license to treating a **WORD** as a reserved word unless it is in a place where a reserved word has to appear.
- The statement is equivalent to requiring that when the LR(1) lookahead set contains exactly one reserved word, it must be recognized if it is present. (Here ''LR(1)'' refers to the theoretical concepts, not to any real parser generator.)

  For example, in the construct below, and when the parser is at the point marked with `'^'`, the only next legal token is **then** (this follows directly from the grammar rules):

  ```
  if if...fi then ... fi
             ^
  ```

  At that point, the **then** must be recognized as a reserved word.

  (Depending on the parser generator actually used, ''extra'' reserved words may be in some lookahead sets. It does not really matter if they are recognized, or even if any possible reserved word is recognized in that state, because if it is recognized and is not in the (theoretical) LR(1) lookahead set, an error is ultimately detected. In the example above, if some other reserved word (for example, **while**) is also recognized, an error occurs later.

  This is approximately equivalent to saying that reserved words are recognized after other reserved words (because it is after a reserved word that this condition occurs), but avoids the ''except for ...'' list that would be required for **case**, **for**, and so on. (Reserved words are of course recognized anywhere a *simple_command* can appear, as well. Other rules take care of the special cases of non-recognition, such as rule (4) for **case** statements.)

Note that the body of here-documents are handled by token recognition (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.3, Token Recognition) and do not appear in the grammar directly. (However, the here-document I/O redirection operator is handled as part of the grammar.)

The start symbol of the grammar (**complete_command**) represents either input from the command line or a shell script. It is repeatedly applied by the interpreter to its input and represents a single ''chunk'' of that input as seen by the interpreter.

C.2.10.1 *Shell Grammar Lexical Conventions*

There is no additional rationale provided for this section.

C.2.10.2 *Shell Grammar Rules*

There is no additional rationale provided for this section.

C.2.11 Signals and Error Handling

There is no additional rationale provided for this section.

C.2.12 Shell Execution Environment

Some implementations have implemented the last stage of a pipeline in the current environment so that commands such as:

```
command | read foo
```

set variable **foo** in the current environment. This extension is allowed, but not required; therefore, a shell programmer should consider a pipeline to be in a subshell environment, but not depend on it.

In early proposals, the description of execution environment failed to mention that each command in a multiple command pipeline could be in a subshell execution environment. For compatibility with some historical shells, the wording was phrased to allow an implementation to place any or all commands of a pipeline in the current environment. However, this means that a POSIX application must assume each command is in a subshell environment, but not depend on it.

The wording about shell scripts is meant to convey the fact that describing ‘‘trap actions’’ can only be understood in the context of the shell command language. Outside of this context, such as in a C-language program, signals are the operative condition, not traps.

C.2.13 Pattern Matching Notation

Pattern matching is a simpler concept and has a simpler syntax than REs, as the former is generally used for the manipulation of filenames, which are relatively simple collections of characters, while the latter is generally used to manipulate arbitrary text strings of potentially greater complexity. However, some of the basic concepts are the same, so this section points liberally to the detailed descriptions in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 9, Regular Expressions.

C.2.13.1 *Patterns Matching a Single Character*

Both quoting and escaping are described here because pattern matching must work in three separate circumstances:

1. Calling directly upon the shell, such as in pathname expansion or in a **case** statement. All of the following match the string or file **abc**:

   ```
   abc "abc" a"b"c a\bc a[b]c a["b"]c a[\b]c a["\b"]c a?c a*c
   ```

   The following do not:

   ```
   "a?c" a\*c a\[b]c
   ```

2. Calling a utility or function without going through a shell, as described for *find* and the *fnmatch*( ) function defined in the System Interfaces volume of IEEE Std 1003.1-200x.

3. Calling utilities such as *find*, *cpio*, *tar*, or *pax* through the shell command line. In this case, shell quote removal is performed before the utility sees the argument. For example, in:

   ```
   find /bin −name "e\c[\h]o" −print
   ```

   after quote removal, the backslashes are presented to *find* and it treats them as escape characters. Both precede ordinary characters, so the *c* and *h* represent themselves and *echo* would be found on many historical systems (that have it in **/bin**). To find a filename that contained shell special characters or pattern characters, both quoting and escaping are required, such as:

   ```
   pax −r ... "*a\(\?"
   ```

   to extract a filename ending with `"a(?"`.

Conforming applications are required to quote or escape the shell special characters (sometimes called metacharacters). If used without this protection, syntax errors can result or implementation extensions can be triggered. For example, the KornShell supports a series of extensions based on parentheses in patterns.

The restriction on a circumflex in a bracket expression is to allow implementations that support pattern matching using the circumflex as the negation character in addition to the exclamation mark. A conforming application must use something like `"[\^!]"` to match either character.

C.2.13.2 Patterns Matching Multiple Characters

Since each asterisk matches zero or more occurrences, the patterns `"a*b"` and `"a**b"` have identical functionality.

Examples

`a[bc]`	Matches the strings `"ab"` and `"ac"`.
`a*d`	Matches the strings `"ad"`, `"abd"`, and `"abcd"`, but not the string `"abc"`.
`a*d*`	Matches the strings `"ad"`, `"abcd"`, `"abcdef"`, `"aaaad"`, and `"adddd"`.
`*a*d`	Matches the strings `"ad"`, `"abcd"`, `"efabcd"`, `"aaaad"`, and `"adddd"`.

C.2.13.3 Patterns Used for Filename Expansion

The caveat about a slash within a bracket expression is derived from historical practice. The pattern `"a[b/c]d"` does not match such pathnames as **abd** or **a/d**. On some implementations (including those conforming to the Single UNIX Specification), it matched a pathname of literally `"a[b/c]d"`. On other systems, it produced an undefined condition (an unescaped `'['` used outside a bracket expression). In this version, the XSI behavior is now required.

Filenames beginning with a period historically have been specially protected from view on UNIX systems. A proposal to allow an explicit period in a bracket expression to match a leading period was considered; it is allowed as an implementation extension, but a conforming application cannot make use of it. If this extension becomes popular in the future, it will be considered for a future version of the Shell and Utilities volume of IEEE Std 1003.1-200x.

Historical systems have varied in their permissions requirements. To match **f*/bar** has required read permissions on the **f*** directories in the System V shell, but the Shell and Utilities volume of IEEE Std 1003.1-200x, the C shell, and KornShell require only search permissions.

C.2.14 Special Built-In Utilities

See the RATIONALE sections on the individual reference pages.

C.3 Batch Environment Services and Utilities

Scope of the Batch Environment Option

This section summarizes the deliberations of the IEEE P1003.15 (Batch Environment) working group in the development of the Batch Environment option, which covers a set of services and utilities defining a batch processing system.

This informative section contains historical information concerning the contents of the amendment and describes why features were included or discarded by the working group.

History of Batch Systems

The supercomputing technical committee began as a ‘‘Birds Of a Feather’’ (BOF) at the January 1987 Usenix meeting. There was enough general interest to form a supercomputing attachment to the /usr/group working groups. Several subgroups rapidly formed. Of those subgroups, the batch group was the most ambitious. The first early meetings were spent evaluating user needs and existing batch implementations.

To evaluate user needs, individuals from the supercomputing community came and presented their needs. Common requests were flexibility, interoperability, control of resources, and ease-of-use. Backwards-compatibility was not an issue. The working group then evaluated some existing systems. The following different systems were evaluated:

- PROD
- Convex Distributed Batch
- NQS
- CTSS
- MDQS from Ballistics Research Laboratory (BRL)

Finally, NQS was chosen as a model because it satisfied not only the most user requirements, but because it was public domain, already implemented on a variety of hardware platforms, and networked-based.

Historical Implementations of Batch Systems

Deferred processing of work under the control of a scheduler has been a feature of most proprietary operating systems from the earliest days of multi-user systems in order to maximize utilization of the computer.

The arrival of UNIX systems proved to be a dilemma to many hardware providers and users because it did not include the sophisticated batch facilities offered by the proprietary systems. This omission was rectified in 1986 by NASA Ames Research Center who developed the Network Queuing System (NQS) as a portable UNIX application that allowed the routing and processing of batch ‘‘jobs’’ in a network. To encourage its usage, the product was later put into the public domain. It was promptly picked up by UNIX hardware providers, and ported and developed for their respective hardware and UNIX implementations.

Many major vendors, who traditionally offer a batch-dominated environment, ported the public-domain product to their systems, customized it to support the capabilities of their systems, and added many customer-requested features.

Due to the strong hardware provider and customer acceptance of NQS, it was decided to use NQS as the basis for the POSIX Batch Environment amendment in 1987. Other batch systems considered at the time included CTSS, MDQS (a forerunner of NQS from the Ballistics Research Laboratory), and PROD (a Los Alamos Labs development). None were thought to have both the functionality and acceptability of NQS.

NQS Differences from the at utility

The base standard *at* and *batch* utilities are not sufficient to meet the batch processing needs in a supercomputing environment and additional functionality in the areas of resource management, job scheduling, system management, and control of output is required.

Batch Environment Option Definitions

The concept of a batch job is closely related to a session with a session leader. The main difference is that a batch job does not have a controlling terminal. There has been much debate over whether to use the term *request* or *job*. Job was the final choice because of the historical use of this term in the batch environment.

The current definition for job identifiers is not sufficient with the model of destinations. The current definition is:

```
sequence_number.originating_host
```

Using the model of destination, a host may include multiple batch nodes, the location of which is identified uniquely by a name or directory service. If the current definition is used, batch nodes running on the same host would have to coordinate their use of sequence numbers, as sequence numbers are assigned by the originating host. The alternative is to use the originating batch node name instead of the originating host name.

The reasons for wishing to run more than one batch system per host could be the following:

A test and production batch system are maintained on a single host. This is most likely in a development facility, but could also arise when a site is moving from one version to another. The new batch system could be installed as a test version that is completely separate from the production batch system, so that problems can be isolated to the test system. Requiring the batch nodes to coordinate their use of sequence numbers creates a dependency between the two nodes, and that defeats the purpose of running two nodes.

A site has multiple departments using a single host, with different management policies. An example of contention might be in job selection algorithms. One group might want a FIFO type of selection, while another group wishes to use a more complex algorithm based on resource availability. Again, requiring the batch nodes to coordinate is an unnecessary binding.

The proposal eventually accepted was to replace originating host with originating batch node. This supplies sufficient granularity to ensure unique job identifiers. If more than one batch node is on a particular host, they each have their own unique name.

The queue portion of a destination is not part of the job identifier as these are not required to be unique between batch nodes. For instance, two batch nodes may both have queues called small, medium, and large. It is only the batch node name that is uniquely identifiable throughout the batch system. The queue name has no additional function in this context.

Assume there are three batch nodes, each of which has its own name server. On batch node one, there are no queues. On batch node two, there are fifty queues. On batch node three, there are forty queues. The system administrator for batch node one does not have to configure queues, because there are none implemented. However, if a user wishes to send a job to either batch node two or three, the system administrator for batch node one must configure a destination that maps to the appropriate batch node and queue. If every queue is to be made accessible from batch node one, the system administrator has to configure ninety destinations.

To avoid requiring this, there should be a mechanism to allow a user to separate the destination into a batch node name and a queue name. Then, an implementation that is configured to get to all the batch nodes does not need any more configuration to allow a user to get to all of the queues on all of the batch nodes. The node name is used to locate the batch node, while the queue name is sent unchanged to that batch node.

The following are requirements that a destination identifier must be capable of providing:

- The ability to direct a job to a queue in a particular batch node.
- The ability to direct a job to a particular batch node.
- The ability to group at a higher level than just one queue. This includes grouping similar queues across multiple batch nodes (this is a pipe queue today).
- The ability to group batch nodes. This allows a user to submit a job to a group name with no knowledge of the batch node configuration. This also provides aliasing as a special case. Aliasing is a group containing only one batch node name. The group name is the alias.

In addition, the administrator has the following requirements:

- The ability to control access to the queues.
- The ability to control access to the batch nodes.
- The ability to control access to groups of queues (pipe queues).
- The ability to configure retry time intervals and durations.

The requirements of the user are met by destination as explained in the following:

The user has the ability to specify a queue name, which is known only to the batch node specified. There is no configuration of these queues required on the submitting node.

The user has the ability to specify a batch node whose name is network-unique. The configuration required is that the batch node be defined as an application, just as other applications such as FTP are configured.

Once a job reaches a queue, it can again become a user of the batch system. The batch node can choose to send the job to another batch node or queue or both. In other words, the routing is at an application level, and it is up to the batch system to choose where the job will be sent. Configuration is up to the batch node where the queue resides. This provides grouping of queues across batch nodes or within a batch node. The user submits the job to a queue, which by definition routes the job to other queues or nodes or both.

A node name may be given to a naming service, which returns multiple addresses as opposed to just one. This provides grouping at a batch node level. This is a local issue, meaning that the batch node must choose only one of these addresses. The list of addresses is not sent with the job, and once the job is accepted on another node, there is no connection between the list and the job. The requirements of the administrator are met by destination as explained in the following:

The control of queues is a batch system issue, and will be done using the batch administrative utilities.

The control of nodes is a network issue, and will be done through whatever network facilities are available.

The control of access to groups of queues (pipe queues) is covered by the control of any other queue. The fact that the job may then be sent to another destination is not relevant.

The propagation of a job across more than one point-to-point connection was dropped because of its complexity and because all of the issues arising from this capability could not be resolved. It could be provided as additional functionality at some time in the future.

The addition of *network* as a defined term was done to clarify the difference between a network of batch nodes as opposed to a network of hosts. A network of batch nodes is referred to as a batch system. The network refers to the actual host configuration. A single host may have multiple batch nodes.

In the absence of a standard network naming convention, this option establishes its own convention for the sake of consistency and expediency. This is subject to change, should a future working group develop a standard naming convention for network pathnames.

C.3.1 Batch General Concepts

During the development of the Batch Environment option, a number of topics were discussed at length which influenced the wording of the normative text but could not be included in the final text. The following items are some of the most significant terms and concepts of those discussed:

- Small and Consistent Command Set

  Often, conventional utilities from UNIX systems have a very complicated utility syntax and usage. This can often result in confusion and errors when trying to use them. The Batch Environment option utility set, on the other hand, has been paired to a small set of robust utilities with an orthogonal calling sequence.

- Checkpoint/Restart

  This feature permits an already executing process to checkpoint or save its contents. Some implementations permit this at both the batch utility level; for example, checkpointing this job upon its abnormal termination or from within the job itself via a system call. Support of checkpoint/restart is optional. A conscious, careful effort was made to make the *qsub* and *qmgr* utilities consistently refer to checkpoint/restart as optional functionality.

- Rerunability

  When a user submits a job for batch processing, they can designate it ''rerunnable'' in that it will automatically resume execution from the start of the job if the machine on which it was executing crashes for some reason. The decision on whether the job will be rerun or not is entirely up to the submitter of the job and no decisions will be made within the batch system. A job that is rerunnable and has been submitted with the proper checkpoint/restart switch will first be checkpointed and execution begun from that point. Furthermore, use of the implementation-defined checkpoint/restart feature will be not be defined in this context.

- Error Codes

  All utilities exit with error status zero (0) if successful, one (1) if a user error occurred, and two (2) for an internal Batch Environment option error.

- Level of Portability

  Portability is specified at both the user, operator, and administrator levels. A conforming batch implementation prevents identical functionality and behavior at all these levels. Additionally, portable batch shell scripts with embedded Batch Environment option utilities adds an additional level of portability.

- Resource Specification

  A small set of globally understood resources, such as memory and CPU time, is specified. All conforming batch implementations are able to process them in a manner consistent with the yet-to-be-developed resource management model. Resources not in this amendment set are ignored and passed along as part of the argument stream of the utility.

- Queue Position

  Queue position is the place a job occupies in a queue. It is dependent on a variety of factors such as submission time and priority. Since priority may be affected by the implementation of fair share scheduling, the definition of queue position is implementation-defined.

- Queue ID

  A numerical queue ID is an external requirement for purposes of accounting. The identification number was chosen over queue name for processing convenience.

- Job ID

  A common notion of ‘‘jobs’’ is a collection of processes whose process group cannot be altered and is used for resource management and accounting. This concept is implementation-defined and, as such, has been omitted from the batch amendment.

- Bytes *versus* Words

  Except for one case, bytes are used as the standard unit for memory size. Furthermore, the definition of a word varies from machine to machine. Therefore, bytes will be the default unit of memory size.

- Regular Expressions

  The standard definition of regular expressions is much too broad to be used in the batch utility syntax. All that is needed is a simple concept of ‘‘all’’; for example, delete all my jobs from the named queue. For this reason, regular expressions have been eliminated from the batch amendment.

- Display Privacy

  How much data should be displayed locally through functions? Local policy dictates the amount of privacy. Library functions must be used to create and enforce local policy. Network and local *qstat*s must reflect the policy of the server machine.

- Remote Host Naming Convention

  It was decided that host names would be a maximum of 255 characters in length, with at most 15 characters being shown in displays. The 255 character limit was chosen because it is consistent with BSD. The 15-character limit was an arbitrary decision.

- Network Administration

  Network administration is important, but is outside the scope of the batch amendment. Network administration could done with *rsh*. However, authentication becomes two-sided.

- Network Administration Philosophy

  Keep it simple. Centralized management should be possible. For example, Los Alamos needs a dumb set of CPUs to be managed by a central system *versus* several independently-managed systems as is the general case for the Batch Environment option.

- Operator Utility Defaults (that is, Default Host, User, Account, and so on)

  It was decided that usability would override orthogonality and syntactic consistency.

- The Batch System Manager and Operator Distinction

  The distinction between manager and operator is that operators can only control the flow of jobs. A manager can alter the batch system configuration in addition to job flow. POSIX makes a distinction between user and system administrator but goes no further. The concepts of manager and operator privileges fall under local policy. The distinction between manager and operator is historical in batch environments, and the Batch Environment option has continued that distinction.

- The Batch System Administrator

  An administrator is equivalent to a batch system manager.

C.3.2 Batch Services

This rationale is provided as informative rather than normative text, to avoid placing requirements on implementors regarding the use of symbolic constants, but at the same time to give implementors a preferred practice for assigning values to these constants to promote interoperability.

The *Checkpoint* and *Minimum_Cpu_Interval* attributes induce a variety of behavior depending upon their values. Some jobs cannot or should not be checkpointed. Other users will simply need to ensure job continuation across planned downtimes; for example, scheduled preventive maintenance. For users consuming expensive resources, or for jobs that run longer than the mean time between failures, however, periodic checkpointing may be essential. However, system administrators must be able to set minimum checkpoint intervals on a queue-by-queue basis to guard against; for example, naive users specifying interval values too small on memory intensive jobs. Otherwise, system overhead would adversely affect performance.

The use of symbolic constants, such as NO_CHECKPOINT, was introduced to lend a degree of formalism and portability to this option.

Support for checkpointing is optional for servers. However, clients must provide for the −**c** option, since in a distributed environment the job may run on a server that does provide such support, even if the host of the client does not support the checkpoint feature.

If the user does not specify the −**c** option, the default action is left unspecified by this option. Some implementations may wish to do checkpointing by default; others may wish to checkpoint only under an explicit request from the user.

The *Priority* attribute has been made non-optional. All clients already had been required to support the −**p** option. The concept of prioritization is common in historical implementations. The default priority is left to the server to establish.

The *Hold_Types* attribute has been modified to allow for implementation-defined hold types to be passed to a batch server.

It was the intent of the IEEE P1003.15 working group to mandate the support for the *Resource_List* attribute in this option by referring to another amendment, specifically P1003.1a. However, during the development of P1003.1a this was excluded. As such this requirement has been removed from the normative text.

The *Shell_Path* attribute has been modified to accept a list of shell paths that are associated with a host. The name of the attribute has been changed to *Shell_Path_List*.

C.3.3 Common Behavior for Batch Environment Utilities

This section was defined to meet the goal of a ‘‘Small and Consistent Command Set’’ for this option.

C.4 Utilities

For the utilities included in IEEE Std 1003.1-200x, see the RATIONALE sections on the individual reference pages.

Exclusion of Utilities

The set of utilities contained in IEEE Std 1003.1-200x is drawn from the base documents, with one addition: the *c99* utility. This section contains rationale for some of the deliberations that led to this set of utilities, and why certain utilities were excluded.

Many utilities were evaluated by the standard developers; more historical utilities were excluded from the base documents than included. The following list contains many common UNIX system utilities that were not included as mandatory utilities, in the UPE, in the XSI extension, or in one of the software development groups. It is logistically difficult for this rationale to distribute correctly the reasons for not including a utility among the various utility options. Therefore, this section covers the reasons for all utilities not included in IEEE Std 1003.1-200x.

This rationale is limited to a discussion of only those utilities actively or indirectly evaluated by the standard developers of the base documents, rather than the list of all known UNIX utilities from all its variants.

adb	The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool. Furthermore, many useful aspects of *adb* are very hardware-specific.
as	Assemblers are hardware-specific and are included implicitly as part of the compilers in IEEE Std 1003.1-200x.
banner	The only known use of this command is as part of the *lp* printer header pages. It was decided that the format of the header is implementation-defined, so this utility is superfluous to application portability.
calendar	This reminder service program is not useful to conforming applications.
cancel	The *lp* (line printer spooling) system specified is the most basic possible and did not need this level of application control.
chroot	This is primarily of administrative use, requiring superuser privileges.
col	No utilities defined in IEEE Std 1003.1-200x produce output requiring such a filter. The *nroff* text formatter is present on many historical systems and will continue to remain as an extension; *col* is expected to be shipped by all the systems that ship *nroff*.
cpio	This has been replaced by *pax*, for reasons explained in the rationale for that utility.
cpp	This is subsumed by *c99*.
cu	This utility is terminal-oriented and is not useful from shell scripts or typical application programs.

dc	The functionality of this utility can be provided by the *bc* utility; *bc* was selected because it was easier to use and had superior functionality. Although the historical versions of *bc* are implemented using *dc* as a base, IEEE Std 1003.1-200x prescribes the interface and not the underlying mechanism used to implement it.
dircmp	Although a useful concept, the historical output of this directory comparison program is not suitable for processing in application programs. Also, the *diff* **−r** command gives equivalent functionality.
dis	Disassemblers are hardware-specific.
emacs	The community of *emacs* editing enthusiasts was adamant that the full *emacs* editor not be included in the base documents because they were concerned that an attempt to standardize this very powerful environment would encourage vendors to ship versions conforming strictly to the standard, but lacking the extensibility required by the community. The author of the original *emacs* program also expressed his desire to omit the program. Furthermore, there were a number of historical UNIX systems that did not include *emacs*, or included it without supporting it, but there were very few that did not include and support *vi*.
ld	This is subsumed by *c99*.
line	The functionality of *line* can be provided with *read*.
lint	This technology is partially subsumed by *c99*. It is also hard to specify the degree of checking for possible error conditions in programs in any compiler, and specifying what *lint* would do in these cases is equally difficult. It is fairly easy to specify what a compiler does. It requires specifying the language, what it does with that language, and stating that the interpretation of any incorrect program is unspecified. Unfortunately, any description of *lint* is required to specify what to do with erroneous programs. Since the number of possible errors and questionable programming practices is infinite, one cannot require *lint* to detect all errors of any given class. Additionally, some vendors complained that since many compilers are distributed in a binary form without a *lint* facility (because the ISO C standard does not require one), implementing the standard as a stand-alone product will be much harder. Rather than being able to build upon a standard compiler component (simply by providing *c99* as an interface), source to that compiler would most likely need to be modified to provide the *lint* functionality. This was considered a major burden on system providers for a very small gain to developers (users).
login	This utility is terminal-oriented and is not useful from shell scripts or typical application programs.
lorder	This utility is an aid in creating an implementation-defined detail of object libraries that the standard developers did not feel required standardization.
lpstat	The *lp* system specified is the most basic possible and did not need this level of application control.
mail	This utility was omitted in favor of *mailx* because there was a considerable functionality overlap between the two.
mknod	This was omitted in favor of *mkfifo*, as *mknod* has too many implementation-defined functions.

news	This utility is terminal-oriented and is not useful from shell scripts or typical application programs.
pack	This compression program was considered inferior to *compress*.
passwd	This utility was proposed in a historical draft of the base documents but met with too many objections to be included. There were various reasons: • Changing a password should not be viewed as a command, but as part of the login sequence. Changing a password should only be done while a trusted path is in effect. • Even though the text in early drafts was intended to allow a variety of implementations to conform, the security policy for one site may differ from another site running with identical hardware and software. One site might use password authentication while the other did not. Vendors could not supply a *passwd* utility that would conform to IEEE Std 1003.1-200x for all sites using their system. • This is really a subject for a system administration working group or a security working group.
pcat	This compression program was considered inferior to *zcat*.
pg	This duplicated many of the features of the *more* pager, which was preferred by the standard developers.
prof	The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.
RCS	RCS was originally considered as part of a version control utilities portion of the scope. However, this aspect was abandoned by the standard developers. SCCS is now included as an optional part of the XSI extension.
red	Restricted editor. This was not considered by the standard developers because it never provided the level of security restriction required.
rsh	Restricted shell. This was not considered by the standard developers because it does not provide the level of security restriction that is implied by historical documentation.
sdb	The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool. Furthermore, some useful aspects of *sdb* are very hardware-specific.
sdiff	The ‘‘side-by-side *diff*’’ utility from System V was omitted because it is used infrequently, and even less so by conforming applications. Despite being in System V, it is not in the SVID or XPG.
shar	Any of the numerous ‘‘shell archivers’’ were excluded because they did not meet the requirement of existing practice.
shl	This utility is terminal-oriented and is not useful from shell scripts or typical application programs. The job control aspects of the shell command language are generally more useful.
size	The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications.

	This utility is primarily a debugging tool.
spell	This utility is not useful from shell scripts or typical application programs. The *spell* utility was considered, but was omitted because there is no known technology that can be used to make it recognize general language for user-specified input without providing a complete dictionary along with the input file.
su	This utility is not useful from shell scripts or typical application programs. (There was also sentiment to avoid security-related utilities.)
sum	This utility was renamed *cksum*.
tar	This has been replaced by *pax*, for reasons explained in the rationale for that utility.
tsort	This utility is an aid in creating an implementation-defined detail of object libraries that the standard developers did not feel required standardization.
unpack	This compression program was considered inferior to *uncompress*.
wall	This utility is terminal-oriented and is not useful from shell scripts or typical application programs. It is generally used only by system administrators.

Rationale (Informative)

Part D:

Portability Considerations

The Open Group
The Institute of Electrical and Electronics Engineers, Inc.

Appendix D

Portability Considerations (Informative)

This section contains information to satisfy various international requirements:

- Section D.1 describes perceived user requirements.
- Section D.2 (on page 3558) indicates how the facilities of IEEE Std 1003.1-200x satisfy those requirements.
- Section D.3 (on page 3565) offers guidance to writers of profiles on how the configurable options, limits, and optional behavior of IEEE Std 1003.1-200x should be cited in profiles.

D.1 User Requirements

This section describes the user requirements that were perceived by the developers of IEEE Std 1003.1-200x. The primary source for these requirements was an analysis of historical practice in widespread use, as typified by the base documents listed in Section A.1.1 (on page 3293).

IEEE Std 1003.1-200x addresses the needs of users requiring open systems solutions for source code portability of applications. It currently addresses users requiring open systems solutions for source-code portability of applications involving multi-programming and process management (creating processes, signaling, and so on); access to files and directories in a hierarchy of file systems (opening, reading, writing, deleting files, and so on); access to asynchronous communications ports and other special devices; access to information about other users of the system; facilities supporting applications requiring bounded (realtime) response.

The following users are identified for IEEE Std 1003.1-200x:

- Those employing applications written in high-level languages, such as C, Ada, or FORTRAN.
- Users who desire conforming applications that do not necessarily require the characteristics of high-level languages (for example, the speed of execution of compiled languages or the relative security of source code intellectual property inherent in the compilation process).
- Users who desire conforming applications that can be developed quickly and can be modified readily without the use of compilers and other system components that may be unavailable on small systems or those without special application development capabilities.
- Users who interact with a system to achieve general-purpose time-sharing capabilities common to most business or government offices or academic environments: editing, filing, inter-user communications, printing, and so on.
- Users who develop applications for POSIX-conformant systems.
- Users who develop applications for UNIX systems.

An acknowledged restriction on applicable users is that they are limited to the group of individuals who are familiar with the style of interaction characteristic of historically-derived systems based on one of the UNIX operating systems (as opposed to other historical systems with different models, such as MS/DOS, Macintosh, VMS, MVS, and so on). Typical users would include program developers, engineers, or general-purpose time-sharing users.

The requirements of users of IEEE Std 1003.1-200x can be summarized as a single goal: *application source portability*. The requirements of the user are stated in terms of the requirements of portability of applications. This in turn becomes a requirement for a standardized set of syntax and semantics for operations commonly found on many operating systems.

The following sections list the perceived requirements for application portability.

D.1.1 Configuration Interrogation

An application must be able to determine whether and how certain optional features are provided and to identify the system upon which it is running, so that it may appropriately adapt to its environment.

Applications must have sufficient information to adapt to varying behaviors of the system.

D.1.2 Process Management

An application must be able to manage itself, either as a single process or as multiple processes. Applications must be able to manage other processes when appropriate.

Applications must be able to identify, control, create, and delete processes, and there must be communication of information between processes and to and from the system.

Applications must be able to use multiple flows of control with a process (threads) and synchronize operations between these flows of control.

D.1.3 Access to Data

Applications must be able to operate on the data stored on the system, access it, and transmit it to other applications. Information must have protection from unauthorized or accidental access or modification.

D.1.4 Access to the Environment

Applications must be able to access the external environment to communicate their input and results.

D.1.5 Access to Determinism and Performance Enhancements

Applications must have sufficient control of resource allocation to ensure the timeliness of interactions with external objects.

D.1.6 Operating System-Dependent Profile

The capabilities of the operating system may make certain optional characteristics of the base language in effect no longer optional, and this should be specified.

D.1.7 I/O Interaction

The interaction between the C language I/O subsystem (*stdio*) and the I/O subsystem of IEEE Std 1003.1-200x must be specified.

D.1.8 Internationalization Interaction

The effects of the environment of IEEE Std 1003.1-200x on the internationalization facilities of the C language must be specified.

D.1.9 C-Language Extensions

Certain functions in the C language must be extended to support the additional capabilities provided by IEEE Std 1003.1-200x.

D.1.10 Command Language

Users should be able to define procedures that combine simple tools and/or applications into higher-level components that perform to the specific needs of the user. The user should be able to store, recall, use, and modify these procedures. These procedures should employ a powerful command language that is used for recurring tasks in conforming applications (scripts) in the same way that it is used interactively to accomplish one-time tasks. The language and the utilities that it uses must be consistent between systems to reduce errors and retraining.

D.1.11 Interactive Facilities

Use the system to accomplish individual tasks at an interactive terminal. The interface should be consistent, intuitive, and offer usability enhancements to increase the productivity of terminal users, reduce errors, and minimize retraining costs. Online documentation or usage assistance should be available.

D.1.12 Accomplish Multiple Tasks Simultaneously

Access applications and interactive facilities from a single terminal without requiring serial execution: switch between multiple interactive tasks; schedule one-time or periodic background work; display the status of all work in progress or scheduled; influence the priority scheduling of work, when authorized.

D.1.13 Complex Data Manipulation

Manipulate data in files in complex ways: sort, merge, compare, translate, edit, format, pattern match, select subsets (strings, columns, fields, rows, and so on). These facilities should be available to both conforming applications and interactive users.

D.1.14 File Hierarchy Manipulation

Create, delete, move/rename, copy, backup/archive, and display files and directories. These facilities should be available to both conforming applications and interactive users.

D.1.15 Locale Configuration

Customize applications and interactive sessions for the cultural and language conventions of the user. Employ a wide variety of standard character encodings. These facilities should be available to both conforming applications and interactive users.

D.1.16 Inter-User Communication

Send messages or transfer files to other users on the same system or other systems on a network. These facilities should be available to both conforming applications and interactive users.

D.1.17 System Environment

Display information about the status of the system (activities of users and their interactive and background work, file system utilization, system time, configuration, and presence of optional facilities) and the environment of the user (terminal characteristics, and so on). Inform the system operator/administrator of problems. Control access to user files and other resources.

D.1.18 Printing

Output files on a variety of output device classes, accessing devices on local or network-connected systems. Control (or influence) the formatting, priority scheduling, and output distribution of work. These facilities should be available to both conforming applications and interactive users.

D.1.19 Software Development

Develop (create and manage source files, compile/interpret, debug) portable open systems applications and package them for distribution to, and updating of, other systems.

D.2 Portability Capabilities

This section describes the significant portability capabilities of IEEE Std 1003.1-200x and indicates how the user requirements listed in Section D.1 (on page 3555) are addressed. The capabilities are listed in the same format as the preceding user requirements; they are summarized below:

- Configuration Interrogation
- Process Management
- Access to Data
- Access to the Environment
- Access to Determinism and Performance Enhancements
- Operating System-Dependent Profile
- I/O Interaction
- Internationalization Interaction
- C-Language Extensions
- Command Language
- Interactive Facilities

- Accomplish Multiple Tasks Simultaneously
- Complex Data Manipulation
- File Hierarchy Manipulation
- Locale Configuration
- Inter-User Communication
- System Environment
- Printing
- Software Development

D.2.1 Configuration Interrogation

The *uname*( ) operation provides basic identification of the system. The *sysconf*( ), *pathconf*( ), and *fpathconf*( ) functions and the *getconf* utility provide means to interrogate the implementation to determine how to adapt to the environment in which it is running. These values can be either static (indicating that all instances of the implementation have the same value) or dynamic (indicating that different instances of the implementation have the different values, or that the value may vary for other reasons, such as reconfiguration).

Unsatisfied Requirements

None directly. However, as new areas are added, there will be a need for additional capability in this area.

D.2.2 Process Management

The *fork*( ), *exec* family, and *spawn*( ) functions provide for the creation of new processes or the insertion of new applications into existing processes. The *_Exit*( ), *_exit*( ), *exit*( ), and *abort*( ) functions allow for the termination of a process by itself. The *wait*( ) and *waitpid*( ) functions allow one process to deal with the termination of another.

The *times*( ) function allows for basic measurement of times used by a process. Various functions, including *fstat*( ), *getegid*( ), *geteuid*( ), *getgid*( ), *getgrgid*( ), *getgrnam*( ), *getlogin*( ), *getpid*( ), *getppid*( ), *getpwnam*( ), *getpwuid*( ), *getuid*( ), *lstat*( ), and *stat*( ), provide for access to the identifiers of processes and the identifiers and names of owners of processes (and files).

The various functions operating on environment variables provide for communication of information (primarily user-configurable defaults) from a parent to child processes.

The operations on the current working directory control and interrogate the directory from which relative filename searches start. The *umask*( ) function controls the default protections applied to files created by the process.

The *alarm*( ), *pause*( ), *sleep*( ), *ualarm*( ), and *usleep*( ) operations allow the process to suspend until a timer has expired or to be notified when a period of time has elapsed. The *time*( ) operation interrogates the current time and date.

The signal mechanism provides for communication of events either from other processes or from the environment to the application, and the means for the application to control the effect of these events. The mechanism provides for external termination of a process and for a process to suspend until an event occurs. The mechanism also provides for a value to be associated with an event.

Job control provides a means to group processes and control them as groups, and to control their access to the function between the user and the system (the *controlling terminal*). It also provides the means to suspend and resume processes.

The Process Scheduling option provides control of the scheduling and priority of a process.

The Message Passing option provides a means for interprocess communication involving small amounts of data.

The Memory Management facilities provide control of memory resources and for the sharing of memory. This functionality is mandatory on XSI-conformant systems.

The Threads facilities provide multiple flows of control with a process (threads), synchronization between threads, association of data with threads, and controlled cancelation of threads.

The XSI interprocess communications functionality provide an alternate set of facilities to manipulate semaphores, message queues, and shared memory. These are provided on XSI-conformant systems to support conforming applications developed to run on UNIX systems. |

D.2.3 Access to Data

The *open*( ), *close*( ), *fclose*( ), *fopen*( ), and *pipe*( ) functions provide for access to files and data. Such files may be regular files, interprocess data channels (pipes), or devices. Additional types of objects in the file system are permitted and are being contemplated for standardization.

The *access*( ), *chmod*( ), *chown*( ), *dup*( ), *dup2*( ), *fchmod*( ), *fcntl*( ), *fstat*( ), *ftruncate*( ), *lstat*( ), *readlink*( ), *realpath*( ), *stat*( ), and *utime*( ) functions allow for control and interrogation of file and file-related objects, (including symbolic links) and their ownership, protections, and timestamps.

The *fgetc*( ), *fputc*( ), *fread*( ), *fseek*( ), *fsetpos*( ), *fwrite*( ), *getc*( ), *getch*( ), *lseek*( ), *putchar*( ), *putc*( ), *read*( ), and *write*( ) functions provide for data transfer from the application to files (in all their forms).

The *closedir*( ), *link*( ), *mkdir*( ), *opendir*( ), *readdir*( ), *rename*( ), *rmdir*( ), *rewinddir*( ), and *unlink*( ) functions provide for a complete set of operations on directories. Directories can arbitrarily contain other directories, and a single file can be mentioned in more than one directory.

The file-locking mechanism provides for advisory locking (protection during transactions) of ranges of bytes (in effect, records) in a file.

The *confstr*( ), *fpathconf*( ), *pathconf*( ), and *sysconf*( ) functions provide for enquiry as to the behavior of the system where variability is permitted.

The Synchronized Input and Output option provides for assured commitment of data to media.

The Asynchronous Input and Output option provides for initiation and control of asynchronous data transfers.

D.2.4 Access to the Environment

The operations and types in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 11, General Terminal Interface are provided for access to asynchronous serial devices. The primary intended use for these is the controlling terminal for the application (the interaction point between the user and the system). They are general enough to be used to control any asynchronous serial device. The functions are also general enough to be used with many other device types as a user interface when some emulation is provided.

Less detailed access is provided for other device types, but in many instances an application need not know whether an object in the file system is a device or a regular file to operate correctly.

Unsatisfied Requirements

Detailed control of common device classes, specifically magnetic tape, is not provided.

D.2.5 Bounded (Realtime) Response

The Realtime Signals Extension provides queued signals and the prioritization of the handling of signals. The SCHED_FIFO, SCHED_SPORADIC, and SCHED_RR scheduling policies provide control over processor allocation. The Semaphores option provides high-performance synchronization. The Memory Management functions provide memory locking for control of memory allocation, file mapping for high-performance, and shared memory for high-performance interprocess communication. The Message Passing option provides for interprocess communication without being dependent on shared memory.

The Timers option provides a high resolution function called *nanosleep*( ) with a finer resolution than the *sleep*( ) function.

The Typed Memory Objects option, the Monotonic Clock option, and the Timeouts option provide further facilities for applications to use to obtain predictable bounded response.

D.2.6 Operating System-Dependent Profile

IEEE Std 1003.1-200x makes no distinction between text and binary files. The values of EXIT_SUCCESS and EXIT_FAILURE are further defined.

Unsatisfied Requirements

None known, but the ISO C standard may contain some additional options that could be specified.

D.2.7 I/O Interaction

IEEE Std 1003.1-200x defines how each of the ISO C standard *stdio* functions interact with the POSIX.1 operations, typically specifying the behavior in terms of POSIX.1 operations.

Unsatisfied Requirements

None.

D.2.8 Internationalization Interaction

The IEEE Std 1003.1-200x environment operations provide a means to define the environment for *setlocale*( ) and time functions such as *ctime*( ). The *tzset*( ) function is provided to set time conversion information.

The *nl_langinfo*( ) function is provided as an XSI extension to query locale-specific cultural settings.

Unsatisfied Requirements

None.

D.2.9 C-Language Extensions

The *setjmp*( ) and *longjmp*( ) functions are not defined to be cognizant of the signal masks defined for POSIX.1. The *sigsetjmp*( ) and *siglongjmp*( ) functions are provided to fill this gap.

The *_setjmp*( ) and *_longjmp*( ) functions are provided as XSI extensions to support historic practice.

Unsatisfied Requirements

None.

D.2.10 Command Language

The shell command language, as described in Shell and Utilities volume of IEEE Std 1003.1-200x, Chapter 2, Shell Command Language, is a common language useful in batch scripts, through an API to high-level languages (for the C-Language Binding option, *system*( ) and *popen*( )) and through an interactive terminal (see the *sh* utility). The shell language has many of the characteristics of a high-level language, but it has been designed to be more suitable for user terminal entry and includes interactive debugging facilities. Through the use of pipelining, many complex commands can be constructed from combinations of data filters and other common components. Shell scripts can be created, stored, recalled, and modified by the user with simple editors.

In addition to the basic shell language, the following utilities offer features that simplify and enhance programmatic access to the utilities and provide features normally found only in high-level languages: *basename*, *bc*, *command*, *dirname*, *echo*, *env*, *expr*, *false*, *printf*, *read*, *sleep*, *tee*, *test*, *time**,[2] *true*, *wait*, *xargs*, and all of the special built-in utilities in the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.14, Special Built-In Utilities .

Unsatisfied Requirements

None.

D.2.11 Interactive Facilities

The utilities offer a common style of command-line interface through conformance to the Utility Syntax Guidelines (see the Base Definitions volume of IEEE Std 1003.1-200x, Section 12.2, Utility Syntax Guidelines) and the common utility defaults (see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 1.11, Utility Description Defaults). The *sh* utility offers an interactive command-line history and editing facility. The following utilities in the User Portability Utilities option have been customized for interactive use: *alias*, *ex*, *fc*, *mailx*, *more*, *talk*, *vi*, *unalias*, and *write*; the *man* utility offers online access to system documentation.

---

[2] The utilities listed with an asterisk here and later in this section are present only on systems which support the User Portability Utilities option. There may be further restrictions on the utilities offered with various configuration option combinations; see the individual utility descriptions.

Unsatisfied Requirements

The command line interface to individual utilities is as intuitive and consistent as historical practice allows. Work underway based on graphical user interfaces may be more suitable for novice or occasional users of the system.

D.2.12 Accomplish Multiple Tasks Simultaneously

The shell command language offers background processing through the asynchronous list command form; see the Shell and Utilities volume of IEEE Std 1003.1-200x, Section 2.9, Shell Commands. The *nohup* utility makes background processing more robust and usable. The *kill* utility can terminate background jobs. When the User Portability Utilities option is supported, the following utilities allow manipulation of jobs: *bg*, *fg*, and *jobs*. Also, if the User Portability Utilities option is supported, the following can support periodic job scheduling, control, and display: *at*, *batch*, *crontab*, *nice*, *ps*, and *renice*.

Unsatisfied Requirements

Terminals with multiple windows may be more suitable for some multi-tasking interactive uses than the job control approach in IEEE Std 1003.1-200x. See the comments on graphical user interfaces in Section D.2.11 (on page 3562). The *nice* and *renice* utilities do not necessarily take advantage of complex system scheduling algorithms that are supported by the realtime options within IEEE Std 1003.1-200x.

D.2.13 Complex Data Manipulation

The following utilities address user requirements in this area: *asa*, *awk*, *bc*, *cmp*, *comm*, *csplit**, *cut*, *dd*, *diff*, *ed*, *ex**, *expand**, *expr*, *find*, *fold*, *grep*, *head*, *join*, *od*, *paste*, *pr*, *printf*, *sed*, *sort*, *split**, *tabs**, *tail*, *tr*, *unexpand**, *uniq*, *uudecode**, *uuencode**, and *wc*.

Unsatisfied Requirements

Sophisticated text formatting utilities, such as *troff* or *TeX*, are not included. Standards work in the area of SGML may satisfy this.

D.2.14 File Hierarchy Manipulation

The following utilities address user requirements in this area: *basename*, *cd*, *chgrp*, *chmod*, *chown*, *cksum*, *cp*, *dd*, *df**, *diff*, *dirname*, *du**, *find*, *ls*, *ln*, *mkdir*, *mkfifo*, *mv*, *patch**, *pathchk*, *pax*, *pwd*, *rm*, *rmdir*, *test*, and *touch*.

Unsatisfied Requirements

Some graphical user interfaces offer more intuitive file manager components that allow file manipulation through the use of icons for novice users.

D.2.15 Locale Configuration

The standard utilities are affected by the various *LC_* variables to achieve locale-dependent operation: character classification, collation sequences, regular expressions and shell pattern matching, date and time formats, numeric formatting, and monetary formatting. When the POSIX2_LOCALEDEF option is supported, applications can provide their own locale definition files. The following utilities address user requirements in this area: *date*, *ed*, *ex**, *find*, *grep*, *locale*, *localedef*, *more**, *sed*, *sh*, *sort*, *tr*, *uniq*, and *vi**.

The *iconv*( ), *iconv_close*( ), and *iconv_open*( ) functions are available to allow an application to convert character data between supported character sets.

The *gencat* utility and the *catopen*( ), *catclose*( ), and *catgets*( ) functions for message catalog manipulation are available on XSI-conformant systems.

Unsatisfied Requirements

Some aspects of multi-byte character and state-encoded character encodings have not yet been addressed. The C-language functions, such as *getopt*( ), are generally limited to single-byte characters. The effect of the *LC_MESSAGES* variable on message formats is only suggested at this time.

D.2.16 Inter-User Communication

The following utilities address user requirements in this area: *cksum*, *mailx**, *mesg**, *patch**, *pax*, *talk**, *uudecode**, *uuencode**, *who**, and *write**.

The historical UUCP utilities are included on XSI-conformant systems.

Unsatisfied Requirements

None.

D.2.17 System Environment

The following utilities address user requirements in this area: *chgrp*, *chmod*, *chown*, *df**, *du**, *env*, *getconf*, *id*, *logger*, *logname*, *mesg**, *newgrp**, *ps**, *stty*, *tput**, *tty*, *umask*, *uname*, and *who**.

The *closelog*( ), *openlog*( ), *setlogmask*( ), and *syslog*( ) functions provide System Logging facilities on XSI-conformant systems; these are analogous to the *logger* utility.

Unsatisfied Requirements

None.

D.2.18 Printing

The following utilities address user requirements in this area: *pr* and *lp*.

Unsatisfied Requirements

There are no features to control the formatting or scheduling of the print jobs.

D.2.19 Software Development

The following utilities address user requirements in this area: *ar*, *asa*, *awk*, *c99*, *ctags**, *fort77*, *getconf*, *getopts*, *lex*, *localedef*, *make*, *nm**, *od*, *patch**, *pax*, *strings**, *strip*, *time**, and *yacc*.

The *system*( ), *popen*( ), *pclose*( ), *regcomp*( ), *regexec*( ), *regerror*( ), *regfree*( ), *fnmatch*( ), *getopt*( ), *glob*( ), *globfree*( ), *wordexp*( ), and *wordfree*( ) functions allow C-language programmers to access some of the interfaces used by the utilities, such as argument processing, regular expressions, and pattern matching.

The SCCS source-code control system utilities are available on systems supporting the XSI Development option.

Unsatisfied Requirements

There are no language-specific development tools related to languages other than C and FORTRAN. The C tools are more complete and varied than the FORTRAN tools. There is no data dictionary or other CASE-like development tools.

D.2.20 Future Growth

It is arguable whether or not all functionality to support applications is potentially within the scope of IEEE Std 1003.1-200x. As a simple matter of practicality, it cannot be. Areas such as graphics, application domain-specific functionality, windowing, and so on, should be in unique standards. As such, they are properly ‘‘Unsatisfied Requirements’’ in terms of providing fully conforming applications, but ones which are outside the scope of IEEE Std 1003.1-200x.

However, as the standards evolve, certain functionality once considered ‘‘exotic’’ enough to be part of a separate standard become common enough to be included in a core standard such as this. Realtime and networking, for example, have both moved from separate standards (with much difficult cross-referencing) into IEEE Std 1003.1 over time, and although no specific areas have been identified for inclusion in future revisions, such inclusions seem likely.

D.3 Profiling Considerations

This section offers guidance to writers of profiles on how the configurable options, limits, and optional behavior of IEEE Std 1003.1-200x should be cited in profiles. Profile writers should consult the general guidance in POSIX.0 when writing POSIX Standardized Profiles.

The information in this section is an inclusive list of features that should be considered by profile writers. Subsetting of IEEE Std 1003.1-200x should follow the Base Definitions volume of IEEE Std 1003.1-200x, Section 2.1.5.1, Subprofiling Considerations. A set of profiling options is described in Appendix E (on page 3579).

D.3.1 Configuration Options

There are two set of options suggested by IEEE Std 1003.1-200x: those for POSIX-conforming systems and those for X/Open System Interface (XSI) conformance. The requirements for XSI conformance are documented in the Base Definitions volume of IEEE Std 1003.1-200x and not discussed further here, as they superset the POSIX conformance requirements.

D.3.2 Configuration Options (Shell and Utilities)

There are three broad optional configurations for the Shell and Utilities volume of IEEE Std 1003.1-200x: basic execution system, development system, and user portability interactive system. The options to support these, and other minor configuration options, are listed in the Base Definitions volume of IEEE Std 1003.1-200x, Chapter 2, Conformance. Profile writers should consult the following list and the comments concerning user requirements addressed by various components in Section D.2 (on page 3558).

POSIX2_UPE

The system supports the User Portability Utilities option.

This option is a requirement for a user portability interactive system. It is required frequently except for those systems, such as embedded realtime or dedicated application systems, that support little or no interactive time-sharing work by users or operators. XSI-conformant systems support this option.

POSIX2_SW_DEV

The system supports the Software Development Utilities option.

This option is required by many systems, even those in which actual software development does not occur. The *make* utility, in particular, is required by many application software packages as they are installed onto the system. If POSIX2_C_DEV is supported, POSIX2_SW_DEV is almost a mandatory requirement because of *ar* and *make*.

POSIX2_C_BIND

The system supports the C-Language Bindings option.

This option is required on some implementations developing complex C applications or on any system installing C applications in source form that require the functions in this option. The *system*( ) and *popen*( ) functions, in particular, are widely used by applications; the others are rather more specialized.

POSIX2_C_DEV

The system supports the C-Language Development Utilities option.

This option is required by many systems, even those in which actual C-language software development does not occur. The *c99* utility, in particular, is required by many application software packages as they are installed onto the system. The *lex* and *yacc* utilities are used less frequently.

POSIX2_FORT_DEV

The system supports the FORTRAN Development Utilities option

As with C, this option is needed on any system developing or installing FORTRAN applications in source form.

POSIX2_FORT_RUN

The system supports the FORTRAN Runtime Utilities option.

This option is required for some FORTRAN applications that need the *asa* utility to convert Hollerith printing statement output. It is unknown how frequently this occurs.

POSIX2_LOCALEDEF

The system supports the creation of locales.

This option is needed if applications require their own customized locale definitions to operate. It is presently unknown whether many applications are dependent on this. However, the option is virtually mandatory for systems in which internationalized applications are developed.

XSI-conformant systems support this option.

POSIX2_PBS

The system supports the Batch Environment option.

POSIX2_PBS_ACCOUNTING

The system supports the optional feature of accounting within the Batch Environment option. It will be required in servers that implement the optional feature of accounting.

POSIX2_PBS_CHECKPOINT

The systems supports the optional feature of checkpoint/restart within the Batch Environment option.

POSIX2_PBS_LOCATE

The system supports the optional feature of locating batch jobs within the Batch Environment option.

POSIX2_PBS_MESSAGE

The system supports the optional feature of sending messages to batch jobs within the Batch Environment option.

POSIX2_PBS_TRACK

The system supports the optional feature of tracking batch jobs within the Batch Environment option.

POSIX2_CHAR_TERM

The system supports at least one terminal type capable of all operations described in IEEE Std 1003.1-200x.

On systems with POSIX2_UPE, this option is almost always required. It was developed solely to allow certain specialized vendors and user applications to bypass the requirement for general-purpose asynchronous terminal support. For example, an application and system that was suitable for block-mode terminals, such as IBM 3270s, would not need this option.

XSI-conformant systems support this option.

D.3.3 Configurable Limits

Very few of the limits need to be increased for profiles. No profile can cite lower values.

{POSIX2_BC_BASE_MAX}
{POSIX2_BC_DIM_MAX}
{POSIX2_BC_SCALE_MAX}
{POSIX2_BC_STRING_MAX}

No increase is anticipated for any of these *bc* values, except for very specialized applications involving huge numbers.

{POSIX2_COLL_WEIGHTS_MAX}

Some natural languages with complex collation requirements require an increase from the default 2 to 4; no higher numbers are anticipated.

{POSIX2_EXPR_NEST_MAX}
No increase is anticipated.

{POSIX2_LINE_MAX}
This number is much larger than most historical applications have been able to use. At some future time, applications may be rewritten to take advantage of even larger values.

{POSIX2_RE_DUP_MAX}
No increase is anticipated.

{POSIX2_VERSION}
This is actually not a limit, but a standard version stamp. Generally, a profile should specify Shell and Utilities volume of IEEE Std 1003.1-200x, Chapter 2, Shell Command Language by name in the normative references section, not this value.

D.3.4 Configuration Options (System Interfaces)

{NGROUPS_MAX}
A non-zero value indicates that the implementation supports supplementary groups.

This option is needed where there is a large amount of shared use of files, but where a certain amount of protection is needed. Many profiles[3] are known to require this option; it should only be required if needed, but it should never be prohibited.

_POSIX_ADVISORY_INFO
The system provides advisory information for file management.

This option allows the application to specify advisory information that can be used to achieve better or even deterministic response time in file manager or input and output operations.

_POSIX_ASYNCHRONOUS_IO
The system provides concurrent process execution and input and output transfers.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

_POSIX_BARRIERS
The system supports barrier synchronization.

This option was created to allow efficient synchronization of multiple parallel threads in multi-processor systems in which the operation is supported in part by the hardware architecture.

_POSIX_CHOWN_RESTRICTED
The system restricts the right to ‘‘give away’’ files to other users.

This option should be carefully investigated before it is required. Some applications expect that they can change the ownership of files in this way. It is provided where either security or system account requirements cause this ability to be a problem. It is also known to be specified in many profiles.

---

[3] There are no formally approved profiles of IEEE Std 1003.1-200x at the time of publication; the reference here is to various profiles generated by private bodies or governments.

_POSIX_CLOCK_SELECTION

The system supports the Clock Selection option.

This option allows applications to request a high resolution sleep in order to suspend a thread during a relative time interval, or until an absolute time value, using the desired clock. It also allows the application to select the clock used in a *pthread_cond_timedwait*( ) function call.

_POSIX_CPUTIME

The system supports the Process CPU-Time Clocks option.

This option allows applications to use a new clock that measures the execution times of processes or threads, and the possibility to create timers based upon these clocks, for runtime detection (and treatment) of execution time overruns.

_POSIX_FSYNC

The system supports file synchronization requests.

This option was created to support historical systems that did not provide the feature. Applications that are expecting guaranteed completion of their input and output operations should require the _POSIX_SYNC_IO option. This option should never be prohibited.

XSI-conformant systems support this option.

_POSIX_IPV6

The system supports facilities related to Internet Protocol Version 6 (IPv6).

This option was created to allow systems to transition to IPv6.

_POSIX_JOB_CONTROL

Job control facilities are mandatory in IEEE Std 1003.1-200x.

The option was created primarily to support historical systems that did not provide the feature. Many existing profiles now require it; it should only be required if needed, but it should never be prohibited. Most applications that use it can run when it is not present, although with a degraded level of user convenience.

_POSIX_MAPPED_FILES

The system supports the mapping of regular files into the process address space.

XSI-conformant systems support this option.

Both this option and the Shared Memory Objects option provide shared access to memory objects in the process address space. The functions defined under this option provide the functionality of existing practice for mapping regular files. This functionality was deemed unnecessary, if not inappropriate, for embedded systems applications and, hence, is provided under this option. It should only be required if needed, but it should never be prohibited.

_POSIX_MEMLOCK

The system supports the locking of the address space.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

_POSIX_MEMLOCK_RANGE

The system supports the locking of specific ranges of the address space.

For applications that have well-defined sections that need to be locked and others that do not, IEEE Std 1003.1-200x supports an optional set of functions to lock or unlock a range of process addresses. The following are two reasons for having a means to lock down a specific range:

1. An asynchronous event handler function that must respond to external events in a deterministic manner such that page faults cannot be tolerated
2. An input/output ''buffer'' area that is the target for direct-to-process I/O, and the overhead of implicit locking and unlocking for each I/O call cannot be tolerated

It should only be required if needed, but it should never be prohibited.

_POSIX_MEMORY_PROTECTION
The system supports memory protection.

XSI-conformant systems support this option.

The provision of this option typically imposes additional hardware requirements. It should never be prohibited.

_POSIX_PRIORITIZED_IO
The system provides prioritization for input and output operations.

The use of this option may interfere with the ability of the system to optimize input and output throughput. It should only be required if needed, but it should never be prohibited.

_POSIX_MESSAGE_PASSING
The system supports the passing of messages between processes.

This option was created to support historical systems that did not provide the feature. The functionality adds a high-performance XSI interprocess communication facility for local communication. It should only be required if needed, but it should never be prohibited.

_POSIX_MONOTONIC_CLOCK
The system supports the Monotonic Clock option.

This option allows realtime applications to rely on a monotonically increasing clock that does not jump backwards, and whose value does not change except for the regular ticking of the clock.

_POSIX_PRIORITY_SCHEDULING
The system provides priority-based process scheduling.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which processes that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_REALTIME_SIGNALS
The system provides prioritized, queued signals with associated data values.

This option was created to support historical systems that did not provide the features. It should only be required if needed, but it should never be prohibited.

_POSIX_REGEXP
Support for regular expression facilities are mandatory in IEEE Std 1003.1-200x.

_POSIX_SAVED_IDS
Support for this feature is mandatory in IEEE Std 1003.1-200x.

Certain classes of applications rely on it for proper operation, and there is no alternative short of giving the application root privileges on most implementations that did not provide _POSIX_SAVED_IDS.

_POSIX_SEMAPHORES

The system provides counting semaphores.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

_POSIX_SHARED_MEMORY_OBJECTS

The system supports the mapping of shared memory objects into the process address space.

Both this option and the Memory Mapped Files option provide shared access to memory objects in the process address space. The functions defined under this option provide the functionality of existing practice for shared memory objects. This functionality was deemed appropriate for embedded systems applications and, hence, is provided under this option. It should only be required if needed, but it should never be prohibited.

_POSIX_SHELL

Support for the *sh* utility command line interpreter is mandatory in IEEE Std 1003.1-200x.

_POSIX_SPAWN

The system supports the spawn option.

This option provides applications with an efficient mechanism to spawn execution of a new process.

_POSIX_SPINLOCKS

The system supports spin locks.

This option was created to support a simple and efficient synchronization mechanism for threads executing in multi-processor systems.

_POSIX_SPORADIC_SERVER

The system supports the sporadic server scheduling policy.

This option provides applications with a new scheduling policy for scheduling aperiodic processes or threads in hard realtime applications.

_POSIX_SYNCHRONIZED_IO

The system supports guaranteed file synchronization.

This option was created to support historical systems that did not provide the feature. Applications that are expecting guaranteed completion of their input and output operations should require this option, rather than the File Synchronization option. It should only be required if needed, but it should never be prohibited.

_POSIX_THREADS

The system supports multiple threads of control within a single process.

This option was created to support historical systems that did not provide the feature. Applications written assuming a multi-threaded environment would be expected to require this option. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_ATTR_STACKADDR

The system supports specification of the stack address for a created thread.

Applications may take advantage of support of this option for performance benefits, but dependence on this feature should be minimized. This option should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_ATTR_STACKSIZE

The system supports specification of the stack size for a created thread.

Applications may require this option in order to ensure proper execution, but such usage limits portability and dependence on this feature should be minimized. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_PRIORITY_SCHEDULING

The system provides priority-based thread scheduling.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_THREAD_PRIO_INHERIT

The system provides mutual exclusion operations with priority inheritance.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_THREAD_PRIO_PROTECT

The system supports a priority ceiling emulation protocol for mutual exclusion operations.

Support of this option provides predictable scheduling behavior, allowing applications to determine the order in which threads that are ready to run are granted access to a processor. It should only be required if needed, but it should never be prohibited.

_POSIX_THREAD_PROCESS_SHARED

The system provides shared access among multiple processes to synchronization objects.

This option was created to support historical systems that did not provide the feature. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_SAFE_FUNCTIONS

The system provides thread-safe versions of all of the POSIX.1 functions.

This option is required if the Threads option is supported. This is a separate option because thread-safe functions are useful in implementations providing other mechanisms for concurrency. It should only be required if needed, but it should never be prohibited.

XSI-conformant systems support this option.

_POSIX_THREAD_SPORADIC_SERVER

The system supports the thread sporadic server scheduling policy.

Support for this option provides applications with a new scheduling policy for scheduling aperiodic threads in hard realtime applications.

_POSIX_TIMEOUTS

The system provides timeouts for some blocking services.

This option was created to provide a timeout capability to system services, thus allowing applications to include better error detection, and recovery capabilities.

_POSIX_TIMERS

The system provides higher resolution clocks with multiple timers per process.

This option was created to support historical systems that did not provide the features. This option is appropriate for applications requiring higher resolution timestamps or needing to control the timing of multiple activities. It should only be required if needed, but it should never be prohibited.

_POSIX_TRACE
The system supports the trace option.

This option was created to allow applications to perform tracing.

_POSIX_TRACE_EVENT_FILTER
The system supports the trace event filter option.

This option is dependent on support of the Trace option.

_POSIX_TRACE_INHERIT
The system supports the trace inherit option.

This option is dependent on support of the Trace option.

_POSIX_TRACE_LOG
The system supports the trace log option.

This option is dependent on support of the Trace option.

_POSIX_TYPED_MEMORY_OBJECTS
The system supports typed memory objects.

This option was created to allow realtime applications to access different kinds of physical memory, and allow processes in these applications to share portions of this memory.

D.3.5 Configurable Limits

In general, the configurable limits in the **<limits.h>** header defined in the Base Definitions volume of IEEE Std 1003.1-200x have been set to minimal values; many applications or implementations may require larger values. No profile can cite lower values.

{AIO_LISTIO_MAX}
The current minimum is likely to be inadequate for most applications. It is expected that this value will be increased by profiles requiring support for list input and output operations.

{AIO_MAX}
The current minimum is likely to be inadequate for most applications. It is expected that this value will be increased by profiles requiring support for asynchronous input and output operations.

{AIO_PRIO_DELTA_MAX}
The functionality associated with this limit is needed only by sophisticated applications. It is not expected that this limit would need to be increased under a general-purpose profile.

{ARG_MAX}
The current minimum is likely to need to be increased for profiles, particularly as larger amounts of information are passed through the environment. Many implementations are believed to support larger values.

{CHILD_MAX}
The current minimum is suitable only for systems where a single user is not running applications in parallel. It is significantly too low for any system also requiring windows, and if _POSIX_JOB_CONTROL is specified, it should be raised.

{CLOCKRES_MIN}

It is expected that profiles will require a finer granularity clock, perhaps as fine as 1 µs, represented by a value of 1 000 for this limit.

{DELAYTIMER_MAX}

It is believed that most implementations will provide larger values.

{LINK_MAX}

For most applications and usage, the current minimum is adequate. Many implementations have a much larger value, but this should not be used as a basis for raising the value unless the applications to be used require it.

{LOGIN_NAME_MAX}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{MAX_CANON}

For most purposes, the current minimum is adequate. Unless high-speed burst serial devices are used, it should be left as is.

{MAX_INPUT}

See {MAX_CANON}.

{MQ_OPEN_MAX}

The current minimum should be adequate for most profiles.

{MQ_PRIO_MAX}

The current minimum corresponds to the required number of process scheduling priorities. Many realtime practitioners believe that the number of message priority levels ought to be the same as the number of execution scheduling priorities.

{NAME_MAX}

Many implementations now support larger values, and many applications and users assume that larger names can be used. Many existing profiles also specify a larger value. Specifying this value will reduce the number of conforming implementations, although this might not be a significant consideration over time. Values greater than 255 should not be required.

{NGROUPS_MAX}

The value selected will typically be 8 or larger.

{OPEN_MAX}

The historically common value for this has been 20. Many implementations support larger values. If applications that use larger values are anticipated, an appropriate value should be specified.

{PAGESIZE}

This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{PATH_MAX}

Historically, the minimum has been either 1 024 or indefinite, depending on the implementation. Few applications actually require values larger than 256, but some users may create file hierarchies that must be accessed with longer paths. This value should only be changed if there is a clear requirement.

{PIPE_BUF}

The current minimum is adequate for most applications. Historically, it has been larger. If applications that write single transactions larger than this are anticipated, it should be increased. Applications that write lines of text larger than this probably do not need it increased, as the text line is delimited by a newline.

{POSIX_VERSION}
This is actually not a limit, but a standard version stamp. Generally, a profile should specify IEEE Std 1003.1-200x by a name in the normative references section, not this value.

{PTHREAD_DESTRUCTOR_ITERATIONS}
It is unlikely that applications will need larger values to avoid loss of memory resources.

{PTHREAD_KEYS_MAX}
The current value should be adequate for most profiles.

{PTHREAD_STACK_MIN}
This should not be treated as an actual limit, but as an implementation parameter. No profile should impose a requirement on this value.

{PTHREAD_THREADS_MAX}
It is believed that most implementations will provide larger values.

{RTSIG_MAX}
The current limit was chosen so that the set of POSIX.1 signal numbers can fit within a 32-bit field. It is recognized that most existing implementations define many more signals than are specified in POSIX.1 and, in fact, many implementations have already exceeded 32 signals (including the ‘‘null signal’’). Support of {_POSIX_RTSIG_MAX} additional signals may push some implementations over the single 32-bit word line, but is unlikely to push any implementations that are already over that line beyond the 64 signal line.

{SEM_NSEMS_MAX}
The current value should be adequate for most profiles.

{SEM_VALUE_MAX}
The current value should be adequate for most profiles.

{SSIZE_MAX}
This limit reflects fundamental hardware characteristics (the size of an integer), and should not be specified unless it is clearly required. Extreme care should be taken to assure that any value that might be specified does not unnecessarily eliminate implementations because of accidents of hardware design.

{STREAM_MAX}
This limit is very closely related to {OPEN_MAX}. It should never be larger than {OPEN_MAX}, but could reasonably be smaller for application areas where most files are not accessed through *stdio*. Some implementations may limit {STREAM_MAX} to 20 but allow {OPEN_MAX} to be considerably larger. Such implementations should be allowed for if the applications permit.

{TIMER_MAX}
The current limit should be adequate for most profiles, but it may need to be larger for applications with a large number of asynchronous operations.

{TTY_NAME_MAX}
This is not actually a limit, but an implementation parameter. No profile should impose a requirement on this value.

{TZNAME_MAX}
The minimum has been historically adequate, but if longer timezone names are anticipated (particularly such values as UTC−1), this should be increased.

D.3.6 Optional Behavior

In IEEE Std 1003.1-200x, there are no instances of the terms unspecified, undefined, implementation-defined, or with the verbs ‘‘may’’ or ‘‘need not’’, that the developers of IEEE Std 1003.1-200x anticipate or sanction as suitable for profile or test method citation. All of these are merely warnings to conforming applications to avoid certain areas that can vary from system to system, and even over time on the same system. In many cases, these terms are used explicitly to support extensions, but profiles should not anticipate and require such extensions; future versions of IEEE Std 1003.1-200x may do so.

Rationale (Informative)

Part E:

Subprofiling Considerations

The Open Group
The Institute of Electrical and Electronics Engineers, Inc.

Appendix E

Subprofiling Considerations (Informative)

This section contains further information to satisfy the requirement that the project scope enable subprofiling of IEEE Std 1003.1-200x. The original intent was to have included a set of options similar to the ‘‘Units of Functionality’’ contained in IEEE Std 1003.13-1998. However, as the development of IEEE Std 1003.1-200x continued, the standard developers felt it premature to fix these in normative text. The approach instead has been to include a general requirement in normative text regarding subprofiling and to include an informative section (here) containing a proposed set of subprofiling options.

E.1 Subprofiling Option Groups

The following Option Groups[4] are defined to support profiling. Systems claiming support to IEEE Std 1003.1-200x need not implement these options apart from the requirements stated in the Base Definitions volume of IEEE Std 1003.1-200x, Section 2.1.3, POSIX Conformance. These Option Groups allow profiles to subset the System Interfaces volume of IEEE Std 1003.1-200x by collecting sets of related functions.

POSIX_C_LANG_JUMP: Jump Functions

longjmp( ), *setjmp*( )

POSIX_C_LANG_MATH: Maths Library

acos( ), *acosf*( ), *acosh*( ), *acoshf*( ), *acoshl*( ), *acosl*( ), *asin*( ), *asinf*( ), *asinh*( ), *asinhf*( ), *asinhl*( ), *asinl*( ), *atan*( ), *atan2*( ), *atan2f*( ), *atan2l*( ), *atanf*( ), *atanh*( ), *atanhf*( ), *atanhl*( ), *atanl*( ), *cabs*( ), *cabsf*( ), *cabsl*( ), *cacos*( ), *cacosf*( ), *cacosh*( ), *cacoshf*( ), *cacoshl*( ), *cacosl*( ), *carg*( ), *cargf*( ), *cargl*( ), *casin*( ), *casinf*( ), *casinh*( ), *casinhf*( ), *casinhl*( ), *casinl*( ), *catan*( ), *catanf*( ), *catanh*( ), *catanhf*( ), *catanhl*( ), *catanl*( ), *cbrt*( ), *cbrtf*( ), *cbrtl*( ), *ccos*( ), *ccosf*( ), *ccosh*( ), *ccoshf*( ), *ccoshl*( ), *ccosl*( ), *ceil*( ), *ceilf*( ), *ceill*( ), *cexp*( ), *cexpf*( ), *cexpl*( ), *cimag*( ), *cimagf*( ), *cimagl*( ), *clog*( ), *clogf*( ), *clogl*( ), *conj*( ), *conjf*( ), *conjl*( ), *copysign*( ), *copysignf*( ), *copysignl*( ), *cos*( ), *cosf*( ), *cosh*( ), *coshf*( ), *coshl*( ), *cosl*( ), *cpow*( ), *cpowf*( ), *cpowl*( ), *cproj*( ), *cprojf*( ), *cprojl*( ), *creal*( ), *crealf*( ), *creall*( ), *csin*( ), *csinf*( ), *csinh*( ), *csinhf*( ), *csinhl*( ), *csinl*( ), *csqrt*( ), *csqrtf*( ), *csqrtl*( ), *ctan*( ), *ctanf*( ), *ctanh*( ), *ctanhf*( ), *ctanhl*( ), *ctanl*( ), *erf*( ), *erfc*( ), *erfcf*( ), *erfcl*( ), *erff*( ), *erfl*( ), *exp*( ), *exp2*( ), *exp2f*( ), *exp2l*( ), *expf*( ), *expl*( ), *expm1*( ), *expm1f*( ), *expm1l*( ), *fabs*( ), *fabsf*( ), *fabsl*( ), *fdim*( ), *fdimf*( ), *fdiml*( ), *floor*( ), *floorf*( ), *floorl*( ), *fma*( ), *fmaf*( ), *fmal*( ), *fmax*( ), *fmaxf*( ), *fmaxl*( ), *fmin*( ), *fminf*( ), *fminl*( ), *fmod*( ), *fmodf*( ), *fmodl*( ), *fpclassify*( ), *frexp*( ), *frexpf*( ), *frexpl*( ), *hypot*( ), *hypotf*( ), *hypotl*( ), *ilogb*( ), *ilogbf*( ), *ilogbl*( ), *isfinite*( ), *isgreater*( ), *isgreaterequal*( ), *isinf*( ), *isless*( ), *islessequal*( ), *islessgreater*( ), *isnan*( ), *isnormal*( ), *isunordered*( ), *ldexp*( ), *ldexpf*( ), *ldexpl*( ), *lgamma*( ), *lgammaf*( ), *lgammal*( ), *llrint*( ), *llrintf*( ), *llrintl*( ), *llround*( ), *llroundf*( ), *llroundl*( ), *log*( ), *log10*( ), *log10f*( ), *log10l*( ), *log1p*( ), *log1pf*( ), *log1pl*( ), *log2*( ), *log2f*( ), *log2l*( ), *logb*( ), *logbf*( ), *logbl*( ), *logf*( ), *logl*( ), *lrint*( ), *lrintf*( ), *lrintl*( ), *lround*( ), *lroundf*( ), *lroundl*( ), *modf*( ), *modff*( ), *modfl*( ), *nan*( ), *nanf*( ), *nanl*( ), *nearbyint*( ), *nearbyintf*( ), *nearbyintl*( ), *nextafter*( ), *nextafterf*( ), *nextafterl*( ), *nexttoward*( ), *nexttowardf*( ), *nexttowardl*( ), *pow*( ), *powf*( ), *powl*( ), *remainder*( ), *remainderf*( ), *remainderl*( ), *remquo*( ), *remquof*( ), *remquol*( ), *rint*( ), *rintf*( ), *rintl*( ), *round*( ), *roundf*( ), *roundl*( ), *scalbln*( ), *scalblnf*( ), *scalblnl*( ), *scalbn*( ), *scalbnf*( ), *scalbnl*( ), *signbit*( ), *sin*( ), *sinf*( ), *sinh*( ), *sinhf*( ), *sinhl*( ), *sinl*( ), *sqrt*( ), *sqrtf*( ), *sqrtl*( ), *tan*( ), *tanf*( ),

---

[4] These are equivalent to the Units of Functionality from IEEE Std 1003.13-1998.

tanh( ), tanhf( ), tanhl( ), tanl( ), tgamma( ), tgammaf( ), tgammal( ), trunc( ), truncf( ), truncl( )

POSIX_C_LANG_SUPPORT: General ISO C Library

abs( ), asctime( ), atof( ), atoi( ), atol( ), bsearch( ), calloc( ), ctime( ), difftime( ), div( ), feclearexcept( ), fegetenv( ), fegetexceptflag( ), fegetround( ), feholdexcept( ), feraiseexcept( ), fesetenv( ), fesetexceptflag( ), fesetround( ), fetestexcept( ), feupdateenv( ), free( ), gmtime( ), imaxabs( ), imaxdiv( ), isalnum( ), isalpha( ), isblank( ), iscntrl( ), isdigit( ), isgraph( ), islower( ), isprint( ), ispunct( ), isspace( ), isupper( ), isxdigit( ), labs( ), ldiv( ), llabs( ), lldiv( ), localeconv( ), localtime( ), malloc( ), memchr( ), memcmp( ), memcpy( ), memmove( ), memset( ), mktime( ), qsort( ), rand( ), realloc( ), setlocale( ), snprintf( ), sprintf( ), srand( ), sscanf( ), strcat( ), strchr( ), strcmp( ), strcoll( ), strcpy( ), strcspn( ), strerror( ), strftime( ), strlen( ), strncat( ), strncmp( ), strncpy( ), strpbrk( ), strrchr( ), strspn( ), strstr( ), strtod( ), strtof( ), strtoimax( ), strtok( ), strtol( ), strtold( ), strtoll( ), strtoul( ), strtoull( ), strtoumax( ), strxfrm( ), time( ), tolower( ), toupper( ), tzname, tzset( ), va_arg( ), va_copy( ), va_end( ), va_start( ), vsnprintf( ), vsprintf( ), vsscanf( )

POSIX_C_LANG_SUPPORT_R: Thread-Safe General ISO C Library

asctime_r( ), ctime_r( ), gmtime_r( ), localtime_r( ), rand_r( ), strerror_r( ), strtok_r( )

POSIX_C_LANG_WIDE_CHAR: Wide-Character ISO C Library

btowc( ), iswalnum( ), iswalpha( ), iswblank( ), iswcntrl( ), iswctype( ), iswdigit( ), iswgraph( ), iswlower( ), iswprint( ), iswpunct( ), iswspace( ), iswupper( ), iswxdigit( ), mblen( ), mbrlen( ), mbrtowc( ), mbsinit( ), mbsrtowcs( ), mbstowcs( ), mbtowc( ), swprintf( ), swscanf( ), towctrans( ), towlower( ), towupper( ), vswprintf( ), vswscanf( ), wcrtomb( ), wcscat( ), wcschr( ), wcscmp( ), wcscoll( ), wcscpy( ), wcscspn( ), wcsftime( ), wcslen( ), wcsncat( ), wcsncmp( ), wcsncpy( ), wcspbrk( ), wcsrchr( ), wcsrtombs( ), wcsspn( ), wcsstr( ), wcstod( ), wcstof( ), wcstoimax( ), wcstok( ), wcstol( ), wcstold( ), wcstoll( ), wcstombs( ), wcstoul( ), wcstoull( ), wcstoumax( ), wcsxfrm( ), wctob( ), wctomb( ), wctrans( ), wctype( ), wmemchr( ), wmemcmp( ), wmemcpy( ), wmemmove( ), wmemset( )

POSIX_C_LIB_EXT: General C Library Extension

fnmatch( ), getopt( ), optarg, opterr, optind, optopt

POSIX_DEVICE_IO: Device Input and Output

FD_CLR( ), FD_ISSET( ), FD_SET( ), FD_ZERO( ), clearerr( ), close( ), fclose( ), fdopen( ), feof( ), ferror( ), fflush( ), fgetc( ), fgets( ), fileno( ), fopen( ), fprintf( ), fputc( ), fputs( ), fread( ), freopen( ), fscanf( ), fwrite( ), getc( ), getchar( ), gets( ), open( ), perror( ), printf( ), pselect( ), putc( ), putchar( ), puts( ), read( ), scanf( ), select( ), setbuf( ), setvbuf( ), stderr, stdin, stdout, ungetc( ), vfprintf( ), vfscanf( ), vprintf( ), vscanf( ), write( )

POSIX_DEVICE_SPECIFIC: General Terminal

cfgetispeed( ), cfgetospeed( ), cfsetispeed( ), cfsetospeed( ), ctermid( ), isatty( ), tcdrain( ), tcflow( ), tcflush( ), tcgetattr( ), tcsendbreak( ), tcsetattr( ), ttyname( )

POSIX_DEVICE_SPECIFIC_R: Thread-Safe General Terminal

ttyname_r( )

POSIX_FD_MGMT: File Descriptor Management

dup( ), dup2( ), fcntl( ), fgetpos( ), fseek( ), fseeko( ), fsetpos( ), ftell( ), ftello( ), ftruncate( ), lseek( ), rewind( )

POSIX_FIFO: FIFO

mkfifo( )

POSIX_FILE_ATTRIBUTES: File Attributes

chmod( ), chown( ), fchmod( ), fchown( ), umask( )

POSIX_FILE_LOCKING: Thread-Safe Stdio Locking

flockfile( ), ftrylockfile( ), funlockfile( ), getc_unlocked( ), getchar_unlocked( ), putc_unlocked( ), *putchar_unlocked*( )

POSIX_FILE_SYSTEM: File System
access( ), *chdir*( ), *closedir*( ), *creat*( ), *fpathconf*( ), *fstat*( ), *getcwd*( ), *link*( ), *mkdir*( ), *opendir*( ), *pathconf*( ), *readdir*( ), *remove*( ), *rename*( ), *rewinddir*( ), *rmdir*( ), *stat*( ), *tmpfile*( ), *tmpnam*( ), *unlink*( ), *utime*( )

POSIX_FILE_SYSTEM_EXT: File System Extensions
glob( ), *globfree*( )

POSIX_FILE_SYSTEM_R: Thread-Safe File System
readdir_r( )

POSIX_JOB_CONTROL: Job Control
setpgid( ), *tcgetpgrp*( ), *tcsetpgrp*( )

POSIX_MULTI_PROCESS: Multiple Processes
_Exit( ), *_exit*( ), *assert*( ), *atexit*( ), *clock*( ), *execl*( ), *execle*( ), *execlp*( ), *execv*( ), *execve*( ), *execvp*( ), *exit*( ), *fork*( ), *getpgrp*( ), *getpid*( ), *getppid*( ), *setsid*( ), *sleep*( ), *times*( ), *wait*( ), *waitpid*( )

POSIX_NETWORKING: Networking
accept( ), *bind*( ), *connect*( ), *endhostent*( ), *endnetent*( ), *endprotoent*( ), *endservent*( ), *freeaddrinfo*( ), *gai_strerror*( ), *getaddrinfo*( ), *gethostbyaddr*( ), *gethostbyname*( ), *gethostent*( ), *gethostname*( ), *getnameinfo*( ), *getnetbyaddr*( ), *getnetbyname*( ), *getnetent*( ), *getpeername*( ), *getprotobyname*( ), *getprotobynumber*( ), *getprotoent*( ), *getservbyname*( ), *getservbyport*( ), *getservent*( ), *getsockname*( ), *getsockopt*( ), *h_errno*, *htonl*( ), *htons*( ), *if_freenameindex*( ), *if_indextoname*( ), *if_nameindex*( ), *if_nametoindex*( ), *inet_addr*( ), *inet_ntoa*( ), *inet_ntop*( ), *inet_pton*( ), *listen*( ), *ntohl*( ), *ntohs*( ), *recv*( ), *recvfrom*( ), *recvmsg*( ), *send*( ), *sendmsg*( ), *sendto*( ), *sethostent*( ), *setnetent*( ), *setprotoent*( ), *setservent*( ), *setsockopt*( ), *shutdown*( ), *socket*( ), *socketpair*( )

POSIX_PIPE: Pipe
pipe( )

POSIX_REGEXP: Regular Expressions
regcomp( ), *regerror*( ), *regexec*( ), *regfree*( )

POSIX_SHELL_FUNC: Shell and Utilities
pclose( ), *popen*( ), *system*( ), *wordexp*( ), *wordfree*( )

POSIX_SIGNALS: Signal
abort( ), *alarm*( ), *kill*( ), *pause*( ), *raise*( ), *sigaction*( ), *sigaddset*( ), *sigdelset*( ), *sigemptyset*( ), *sigfillset*( ), *sigismember*( ), *signal*( ), *sigpending*( ), *sigprocmask*( ), *sigsuspend*( ), *sigwait*( )

POSIX_SIGNAL_JUMP: Signal Jump Functions
siglongjmp( ), *sigsetjmp*( )

POSIX_SINGLE_PROCESS: Single Process
confstr( ), *environ*, *errno*, *getenv*( ), *setenv*( ), *sysconf*( ), *uname*( ), *unsetenv*( )

POSIX_SYMBOLIC_LINKS: Symbolic Links
lstat( ), *readlink*( ), *symlink*( )

POSIX_SYSTEM_DATABASE: System Database
getgrgid( ), *getgrnam*( ), *getpwnam*( ), *getpwuid*( )

POSIX_SYSTEM_DATABASE_R: Thread-Safe System Database
getgrgid_r( ), *getgrnam_r*( ), *getpwnam_r*( ), *getpwuid_r*( )

POSIX_USER_GROUPS: User and Group

getegid( ), *geteuid*( ), *getgid*( ), *getgroups*( ), *getlogin*( ), *getuid*( ), *setegid*( ), *seteuid*( ), *setgid*( ), *setuid*( )

POSIX_USER_GROUPS_R: Thread-Safe User and Group

getlogin_r( )

POSIX_WIDE_CHAR_DEVICE_IO: Device Input and Output

fgetwc( ), *fgetws*( ), *fputwc*( ), *fputws*( ), *fwide*( ), *fwprintf*( ), *fwscanf*( ), *getwc*( ), *getwchar*( ), *putwc*( ), *putwchar*( ), *ungetwc*( ), *vfwprintf*( ), *vfwscanf*( ), *vwprintf*( ), *vwscanf*( ), *wprintf*( ), *wscanf*( )

XSI_C_LANG_SUPPORT: XSI General C Library

_tolower( ), *_toupper*( ), *a64l*( ), *daylight*( ), *drand48*( ), *erand48*( ), *ffs*( ), *getcontext*( ), *getdate*( ), *getsubopt*( ), *hcreate*( ), *hdestroy*( ), *hsearch*( ), *iconv*( ), *iconv_close*( ), *iconv_open*( ), *initstate*( ), *insque*( ), *isascii*( ), *jrand48*( ), *l64a*( ), *lcong48*( ), *lfind*( ), *lrand48*( ), *lsearch*( ), *makecontext*( ), *memccpy*( ), *mrand48*( ), *nrand48*( ), *random*( ), *remque*( ), *seed48*( ), *setcontext*( ), *setstate*( ), *signgam*( ), *srand48*( ), *srandom*( ), *strcasecmp*( ), *strdup*( ), *strfmon*( ), *strncasecmp*( ), *strptime*( ), *swab*( ), *swapcontext*( ), *tdelete*( ), *tfind*( ), *timezone*( ), *toascii*( ), *tsearch*( ), *twalk*( )

XSI_DBM: XSI Database Management

dbm_clearerr( ), *dbm_close*( ), *dbm_delete*( ), *dbm_error*( ), *dbm_fetch*( ), *dbm_firstkey*( ), *dbm_nextkey*( ), *dbm_open*( ), *dbm_store*( )

XSI_DEVICE_IO: XSI Device Input and Output

fmtmsg( ), *poll*( ), *pread*( ), *pwrite*( ), *readv*( ), *writev*( )

XSI_DEVICE_SPECIFIC: XSI General Terminal

grantpt( ), *posix_openpt*( ), *ptsname*( ), *unlockpt*( )

XSI_DYNAMIC_LINKING: XSI Dynamic Linking

dlclose( ), *dlerror*( ), *dlopen*( ), *dlsym*( )

XSI_FD_MGMT: XSI File Descriptor Management

truncate( )

XSI_FILE_SYSTEM: XSI File System

basename( ), *dirname*( ), *fchdir*( ), *fstatvfs*( ), *ftw*( ), *getwd*( ), *lchown*( ), *lockf*( ), *mknod*( ), *mkstemp*( ), *mktemp*( ), *nftw*( ), *realpath*( ), *seekdir*( ), *statvfs*( ), *sync*( ), *telldir*( ), *tempnam*( ), *utimes*( )

XSI_I18N: XSI Internationalization

catclose( ), *catgets*( ), *catopen*( ), *nl_langinfo*( )

XSI_IPC: XSI Interprocess Communication

ftok( ), *msgctl*( ), *msgget*( ), *msgrcv*( ), *msgsnd*( ), *semctl*( ), *semget*( ), *semop*( ), *shmat*( ), *shmctl*( ), *shmdt*( ), *shmget*( )

XSI_JOB_CONTROL: XSI Job Control

tcgetsid( )

XSI_JUMP: XSI Jump Functions

_longjmp( ), *_setjmp*( )

XSI_MATH: XSI Maths Library

j0( ), *j1*( ), *jn*( ), *scalb*( ), *y0*( ), *y1*( ), *yn*( )

XSI_MULTI_PROCESS: XSI Multiple Process

getpgid( ), *getpriority*( ), *getrlimit*( ), *getrusage*( ), *getsid*( ), *nice*( ), *setpgrp*( ), *setpriority*( ), *setrlimit*( ), *ulimit*( ), *usleep*( ), *vfork*( ), *waitid*( )

XSI_SIGNALS: XSI Signal
bsd_signal( ), *killpg*( ), *sigaltstack*( ), *sighold*( ), *sigignore*( ), *siginterrupt*( ), *sigpause*( ), *sigrelse*( ), *sigset*( ), *ualarm*( )

XSI_SINGLE_PROCESS: XSI Single Process
ftime( ), *gethostid*( ), *gettimeofday*( ), *putenv*( )

XSI_SYSTEM_DATABASE: XSI System Database
endpwent( ), *getpwent*( ), *setpwent*( )

XSI_SYSTEM_LOGGING: XSI System Logging
closelog( ), *openlog*( ), *setlogmask*( ), *syslog*( )

XSI_THREAD_MUTEX_EXT: XSI Thread Mutex Extensions
pthread_mutexattr_gettype( ), *pthread_mutexattr_settype*( )

XSI_THREADS_EXT: XSI Threads Extensions
pthread_attr_getguardsize( ), *pthread_attr_getstack*( ), *pthread_attr_setguardsize*( ), *pthread_attr_setstack*( ), *pthread_getconcurrency*( ), *pthread_setconcurrency*( )

XSI_TIMERS: XSI Timers
getitimer( ), *setitimer*( )

XSI_USER_GROUPS: XSI User and Group
endgrent( ), *endutxent*( ), *getgrent*( ), *getutxent*( ), *getutxid*( ), *getutxline*( ), *pututxline*( ), *setgrent*( ), *setregid*( ), *setreuid*( ), *setutxent*( )

XSI_WIDE_CHAR: XSI Wide-Character Library
wcswcs( ), *wcswidth*( ), *wcwidth*( )

Index