ISO/IEC JTC 1/SC 22/WG14 **N 9999**

March 30, 2019 **v 1**

Remove conditional “WANT” macros from numbered clauses
proposal for C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

The recent integration of TS 18661-1 has moved the use of “WANT” macros into the main body of the C standard, making the added interfaces optional. We think that this is not optimal, neither for user code nor for implementations, an propose to change that to a set of more straight forward feature test macros for the version of the included headers. Along with that also a long list of names have been imposed to the standard. We propose some mild modifications to reduce the pain of the transition and keep C open for future directions.

1. INTRODUCTION

When it was designed, TS 18661-1 (and follow ups) invented a mechanism that would allow implementations to provide that extension in the concerned headers without imposing a pollution of the user name space for code that was not TS 18661-1 aware. Whereas in that context the approach made complete sense, continuing with the same setting once integrated into ISO/IEC 9899 is not very constructive.

- It makes interfaces optional that shouldn’t be.
- It reduces exposure of the new interfaces to a very restricted set of applications.
- It adds unnecessary complexity to implementations.

On the other hand, adding new mandatory interfaces to standard headers also has its cost, namely the increasing risk of name conflicts with an existing code base. This risk is relatively high for TS 18661-1:

- TS 18661-1 adds about 150 (13%) new interfaces (functions and macros) to the C standard.
- Some of these interfaces use plain English words (**`canonicalize`**), short abbreviations (**`daddl`**) or introduce unusual naming schemes (**`fromfp`**), that have an even higher risk of name conflicts that the usual prefix-oriented additions.

The proposal of this paper is to remove the conditionality of these interfaces by

(1) removing the dependency from the **`__STDC_WANT_IEC_60559_BFP_EXT__`** macro,
(2) by adding version test macros such as `__STDC_FENV_VERSION__` to the headers that undergo changes,
(3) by revisiting some of the naming choices, and
(4) by reserving some identifier prefixes for future use.

2. REMOVING DEPENDENCY FROM `__STDC_WANT_IEC_60559_BFP_EXT__`

The only construct in the standard that would be similar to **`__STDC_WANT_IEC_60559_BFP_EXT__`** is **`__STDC_WANT_LIB_EXT1__`** as it used by Annex K. Since the features of Annex K are optional (testable by **`__STDC_LIB_EXT1__`**) such a macro makes complete sense there, because we don’t want an implementation that has Annex K to pollute the name space of all its users.
For the integration of TS 18661-1 the situation is different. It has mainly (see below) integrated directly into the body of the standard, and there is no reason (or feature test macro) that indicates that the interfaces should be optional. In the contrary, most of them are useful additions that should make coding with floating point data more convenient and numerical algorithms more robust.

There are only a few new interfaces that are not integrated into the body of the standard but into Annex F, where a dependency from __STDC_WANT_IEC_60559_BFP_EXT__ makes perfect sense, namely for the same reasons as mentioned above for Annex K.
Therefore we simply propose

— to move the boilerplate for WANT macros from 7.1.2 (Standard headers) to Annex F.
— to remove the use of __STDC_WANT_IEC_60559_BFP_EXT__ from all numbered clauses, but to keep it in Annex F.

Editorially these two steps are quite easy, and we show their application in the attached diffmarks.

3. ADDING VERSION TEST MACROS

The addition of about 150 new interfaces for a new C version can be quite a burden for large code bases that wish to migrate to C2x. Conflicts will not occur often, but they are likely to occur *somewhere* and should be easy to track and to manage.
Therefore we should provide an easy-to-use tool that allows for user code to control the possible damage, but on the other hand will not impose much of a maintenance burden for implementations either.
Another difficulty that appears when the community moves to a new C standard is the fact that nowadays compilers and C libraries often come from different hands, and thus their synchronization concerning a new standard is not trivial. History has shown that this has been mayor hurdle for early acceptance of new C standards, and that dependency of one single "language" version macro __STDC_VERSION__ is not enough to clarify the situation.
Therefore we propose to use a set of new macros of the form __STDC_*XXXX*_VERSION__.
For example <math.h> sets a new macro __STDC_MATH_VERSION__ to a value greater than 202000L, and users can then test this as follows.

```
#include <math.h>
#if __STDC_MATH_VERSION__ > 202000L
#  error "this␣code␣likes␣to␣daddl,␣fix␣before␣going␣further"
#endif
```

There is already large experience with the use of such version macros for library headers in ISO/IEC 9945, POSIX. There, such macros are defined for major branches of the standard and applications have learned to deal with them to adapt their code to the actual environment.

4. REVISITING SOME OF THE NAMING CHOICES

Many of the new interfaces would better have been introduced with a name prefix, much as other headers did when they were added to the C standard. It seems that this opportunity has been missed, though I think that we still could take a turn and use names such as fp_canonicalize instead of **canonicalize**, fp_add instead of **fadd**, etc.
Where these additions are particularly bad is where they introduce a new naming scheme (without admitting it) that is even contraproductive to a future encapsulation of these interfaces in a type generic function. These are the functions

fromfpf
fromfpl
fromfpxf
fromfpxl
fromfpx
fromfp
strfromd
strfromf
strfroml
ufromfpf
ufromfpl
ufromfpxf
ufromfpxl
ufromfpx
ufromfp

Here the usage of the particle `from` has no precedent in the standard. It is not a good choice because in C conversions do usually not specify the source type of a conversion (it can be deduced from the context) but, if so, the target type. By the naming choice, these interfaces cannot be easily extended to type generic interfaces, since by their nature these should have the source type implicit and the target type of feature explicit.
Therefore we propose to rename these interfaces to names starting with the reserved prefix **`to`**, namely

`tointf`
`tointl`
`tointxf`
`tointxl`
`tointx`
`toint`
`tostrd`
`tostrf`
`tostrl`
`touintf`
`touintl`
`touintxf`
`touintxl`
`touintx`
`touint`

This clears up the type generic interfaces in `<tgmath.h>` (to `toint` and `touint`) and will permit to propose another type generic interface in the sequel, in particular a macro `tostr` for a type generic and safe conversion interface conversion from any base type to a string.

5. RESERVE ACTIVE PREFIXES FOR FUTURE USE

The integration of TS 18661-1 has also shown that four prefixes are actively used for new macro interfaces (namely `DBL_`, `FLT_`, `LDBL_` and **`FP_`**) and should thus not be used by user code. Therefore we propose to reserve these for future use. In addition, we propose also to extend the future use clauses of some other prefixes to the header files were they are actually used.

Appendix: pages with diffmarks of the proposed changes against the March 2019 working draft.

The following page numbers are from the particular snapshot and may vary once the changes are integrated.

- 63 nesting levels of parenthesized declarators within a full declarator
- 63 nesting levels of parenthesized expressions within a full expression
- 63 significant initial characters in an internal identifier or a macro name(each universal character name or extended source character is considered a single character)
- 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[19]
- 4095 external identifiers in one translation unit
- 511 identifiers with block scope declared in one block
- 4095 macro identifiers simultaneously defined in one preprocessing translation unit
- 127 parameters in one function definition
- 127 arguments in one function call
- 127 parameters in one macro definition
- 127 arguments in one macro invocation
- 4095 characters in a logical source line
- 4095 characters in a string literal (after concatenation)
- 65535 bytes in an object (in a hosted environment only)
- 15 nesting levels for **`#include`**d files
- 1023 **`case`** labels for a **`switch`** statement (excluding those for any nested **`switch`** statements)
- 1023 members in a single structure or union
- 1023 enumeration constants in a single enumeration
- 63 levels of nested structure or union definitions in a single member declaration list

5.2.4.2 Numerical limits

1 An implementation is required to document all the limits specified in this subclause, which are specified in the headers `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`.

Forward references: integer types `<stdint.h>` (7.20).

5.2.4.2.1 Sizes of integer types `<limits.h>`

1 The ~~following identifiers are defined only if **`__STDC_WANT_IEC_60559_BFP_EXT__`** is defined as a macro at the point in the source file where is first included:~~

~~**`CHAR_WIDTH`**~~
~~**`SCHAR_WIDTH`**~~
~~**`UCHAR_WIDTH`**~~
~~**`SHRT_WIDTH`**~~
~~**`USHRT_WIDTH`**~~
~~**`INT_WIDTH`**~~
~~**`UINT_WIDTH`**~~
~~**`LONG_WIDTH`**~~
~~**`ULONG_WIDTH`**~~

[19] See "future language directions" (6.11.3).

~~**`LLONG_WIDTH`**~~
~~**`ULLONG_WIDTH`**~~

~~The~~ values given below shall be replaced by constant expressions suitable for use in **`#if`** preprocessing directives. Moreover, except for **`CHAR_BIT`** and **`MB_LEN_MAX`**, and the width-of-type macros, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

— number of bits for smallest object that is not a bit-field (byte)

`CHAR_BIT                     8`

— minimum value for an object of type **`signed char`**

`SCHAR_MIN                 -127 //` $-(2^{7}-1)$

— maximum value for an object of type **`signed char`**

`SCHAR_MAX                 +127 //` $2^{7}-1$

— width of type **`signed char`**

`SCHAR_WIDTH                  8`

— maximum value for an object of type **`unsigned char`**

`UCHAR_MAX                  255 //` $2^{8}-1$

— width of type **`unsigned char`**

`UCHAR_WIDTH                  8`

— minimum value for an object of type **`char`**

`CHAR_MIN             ` *`see below`*

— maximum value for an object of type **`char`**

`CHAR_MAX             ` *`see below`*

— width of type **`char`**

<stdnoreturn.h>
<string.h>
<tgmath.h>
<threads.h>
<time.h>
<uchar.h>
<wchar.h>
<wctype.h>

3 If a file with the same name as one of the above < and > delimited sequences, not provided as part of the implementation, is placed in any of the standard places that are searched for included source files, the behavior is undefined.

4 Standard headers may be included in any order; each may be included more than once in a given scope, with no effect different from being included only once, except that the effect of including <assert.h> depends on the definition of NDEBUG (see 7.2). If used, a header shall be included outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if an identifier is declared or defined in more than one header, the second and subsequent associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion of the header or when any macro defined in the header is expanded.

5 Some standard headers define or declare identifiers ~~contingent on whether certain macros whose names begin with __STDC_WANT_IEC_60559_ and end with _EXT__ are defined (by the user) at the point in the code where the header is first included. Within a preprocessing translation unit,~~ that had not been present in previous versions of this document. To allow implementations and users to adapt to that situation, they also define a version macro for feature test of the form __STDC_*XXXX*_VERSION__ which expands to *yyyymm*L, where XXXX is the all-caps spelling of the ~~same set of such macros shall be defined for the first inclusion of all such headers.~~ corresponding header <xxxx.h>.

6 Any definition of an object-like macro described in this clause or Annex K shall expand to code that is fully protected by parentheses where necessary, so that it groups in an arbitrary expression as if it were a single identifier.

7 Any declaration of a library function shall have external linkage.

8 A summary of the contents of the standard headers is given in Annex B.

Forward references: diagnostics (7.2).

7.1.3 Reserved identifiers

1 Each header declares or defines all identifiers listed in its associated subclause, and optionally declares or defines identifiers listed in its associated future library directions subclause and identifiers which are always reserved either for any use or for use as file scope identifiers.

- All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use, except those identifiers which are lexically identical to keywords.[190]
- All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary and tag name spaces.
- Each macro name in any of the following subclauses (including the future library directions) is reserved for use as specified if any of its associated headers is included; unless explicitly stated otherwise (see 7.1.4).
- All identifiers with external linkage in any of the following subclauses (including the future library directions) and errno are always reserved for use as identifiers with external linkage.[191]
- Each identifier with file scope listed in any of the following subclauses (including the future library directions) is reserved for use as a macro name and as an identifier with file scope in the same name space if any of its associated headers is included.

[190] Allows identifiers spelled with a leading underscore followed by an uppercase letter that match the spelling of a keyword to be used as macro names by the program.

[191] The list of reserved identifiers with external linkage includes math_errhandling, setjmp, va_copy, and va_end.

7.6 Floating-point environment `<fenv.h>`

1 The header `<fenv.h>` defines several macros, and declares types and functions that provide access to the floating-point environment. The *floating-point environment* refers collectively to any floating-point status flags and control modes supported by the implementation.[211] A *floating-point status flag* is a system variable whose value is set (but never cleared) when a *floating-point exception* is raised, which occurs as a side effect of exceptional floating-point arithmetic to provide auxiliary information.[212] A *floating-point control mode* is a system variable whose value may be set by the user to affect the subsequent behavior of floating-point arithmetic.

2 A floating-point control mode may be *constant* (7.6.2) or *dynamic*. The *dynamic floating-point environment* includes the dynamic floating-point control modes and the floating-point status flags.

3 The dynamic floating-point environment has thread storage duration. The initial state for a thread's dynamic floating-point environment is the current state of the dynamic floating-point environment of the thread that creates it at the time of creation.

4 Certain programming conventions support the intended model of use for the dynamic floating-point environment:[213]

— a function call does not alter its caller's floating-point control modes, clear its caller's floating-point status flags, nor depend on the state of its caller's floating-point status flags unless the function is so documented;

— a function call is assumed to require default floating-point control modes, unless its documentation promises otherwise;

— a function call is assumed to have the potential for raising floating-point exceptions, unless its documentation promises otherwise.

5 ~~The following identifiers are defined or declared only if __STDC_WANT_IEC_60559_BFP_EXT__ is defined as a macro at the point in the source file where is first included: **femode_t**~~
~~**FE_DFL_MODE**~~
~~**FE_SNANS_ALWAYS_SIGNAL**~~
~~**fesetexcept**~~
~~**fetestexceptflag**~~
~~**fegetmode**~~
~~**fesetmode**~~
The feature test macro **__STDC_FENV_VERSION__** expands to the token *yyyymm*L.

6 The type

```
        fenv_t
```

represents the entire dynamic floating-point environment.

7 The type

```
        femode_t
```

represents the collection of dynamic floating-point control modes supported by the implementation, including the dynamic rounding direction mode.

8 The type

[211] This header is designed to support the floating-point exception status flags and directed-rounding control modes required by IEC 60559, and other similar floating-point state information. It is also designed to facilitate code portability among all systems.

[212] A floating-point status flag is not an object and can be set more than once within an expression.

[213] With these conventions, a programmer can safely assume default floating-point control modes (or be unaware of them). The responsibilities associated with accessing the floating-point environment fall on the programmer or program that does so explicitly.

`<stdlib.h>`	**atof**, ~~strfromd, strfromf, strfroml,~~ **strtod**, **strtof**, **strtold**, **tostrd**, **tostrf**, **tostrl**
`<wchar.h>`	**wcstod**, **wcstof**, **wcstold**
`<stdio.h>`	**printf** and **scanf** families
`<wchar.h>`	**wprintf** and **wscanf** families

Each `<math.h>` function listed in the table above indicates the family of functions of all supported types (for example, **acosf** and **acosl** as well as **acos**).

5 **NOTE** Constant rounding modes (other than **FE_DYNAMIC**) could be implemented using dynamic rounding modes as illustrated in the following example:

```
{
      #pragma STDC FENV_ROUND direction
      // compiler inserts:
      // #pragma STDC FENV_ACCESS ON
      // int __savedrnd;
      // __savedrnd = __swapround(direction);
      ...  operations affected by constant rounding mode ...
      // compiler inserts:
      // __savedrnd = __swapround(__savedrnd);
      ...  operations not affected by constant rounding mode ...
      // compiler inserts:
      // __savedrnd = __swapround(__savedrnd);
      ...  operations affected by constant rounding mode ...
      // compiler inserts:
      // __swapround(__savedrnd);
}
```

where `__swapround` is defined by:

```
static inline int __swapround(const int new) {
      const int old = fegetround();
      fesetround(new);
      return old;
}
```

7.6.3 Floating-point exceptions

1 The following functions provide access to the floating-point status flags.[222] The **int** input argument for the functions represents a subset of floating-point exceptions, and can be zero or the bitwise OR of one or more floating-point exception macros, for example **FE_OVERFLOW | FE_INEXACT**. For other argument values, the behavior of these functions is undefined.

7.6.3.1 The feclearexcept function

Synopsis

1
```
#include <fenv.h>
int feclearexcept(int excepts);
```

Description

2 The **feclearexcept** function attempts to clear the supported floating-point exceptions represented by its argument.

[222] The functions **fetestexcept**, **feraiseexcept**, and **feclearexcept** support the basic abstraction of flags that are either set or clear. An implementation can endow floating-point status flags with more information — for example, the address of the code which first raised the floating-point exception; the functions **fegetexceptflag** and **fesetexceptflag** deal with the full content of flags.

7.12 Mathematics `<math.h>`

1 The header `<math.h>` declares two types and many mathematical functions and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more **`double`** parameters, a **`double`** return value, or both; and other functions with the same name but with **`f`** and **`l`** suffixes, which are corresponding functions with **`float`** and **`long double`** parameters, return values, or both.[234] Integer arithmetic functions and conversion functions are discussed later.

2 ~~The following identifiers are defined or declared only if __STDC_WANT_IEC_60559_BFP_EXT__ is defined as a macro at the point in the source file where is first included: FP_INT_UPWARD~~
~~FP_INT_DOWNWARD~~
~~FP_INT_TOWARDZERO~~
~~FP_INT_TONEARESTFROMZERO~~
~~FP_INT_TONEAREST~~
~~FP_LLOGB0~~
~~FP_LLOGBNAN~~
~~SNANF~~
~~SNAN~~
~~SNANL~~
~~FP_FAST_FADD~~
~~FP_FAST_FADDL~~
~~FP_FAST_DADDL~~
~~FP_FAST_FSUB~~
~~FP_FAST_FSUBL~~
~~FP_FAST_DSUBL~~
~~FP_FAST_FMUL~~
~~FP_FAST_FMULL~~
~~FP_FAST_DMULL~~
~~FP_FAST_FDIV~~
~~FP_FAST_FDIVL~~
~~FP_FAST_DDIVL~~
~~FP_FAST_FFMA~~
~~FP_FAST_FFMAL~~
~~FP_FAST_DFMAL~~
~~FP_FAST_FSQRT~~
~~FP_FAST_FSQRTL~~
~~FP_FAST_DSQRTL~~
~~iseqsig~~
~~iscanonical~~
~~issignaling~~
~~issubnormal~~
~~iszero~~
~~fromfp~~
~~fromfpf~~
~~fromfpl~~
~~ufromfp~~
~~ufromfpf~~
~~ufromfpl~~
~~fromfpx~~
~~fromfpxf~~
~~fromfpxl~~
~~ufromfpx~~
~~ufromfpxf~~
~~ufromfpxl~~

[234] Particularly on systems with wide expression evaluation, a `<math.h>` function might pass arguments and return values in wider format than the synopsis prototype indicates.

~~roundeven~~
~~roundevenf~~
~~roundevenl~~
~~llogb~~
~~llogbf~~
~~llogbl~~
~~fmaxmag~~
~~fmaxmagf~~
~~fmaxmagl~~
~~fminmag~~
~~fminmagf~~
~~fminmagl~~
~~nextup~~
~~nextupf~~
~~nextupl~~
~~nextdown~~
~~nextdownf~~
~~nextdownl~~
~~fadd~~
~~faddl~~
~~daddl~~
~~fsub~~
~~fsubl~~
~~dsubl~~
~~fmul~~
~~fmull~~
~~dmull~~
~~fdiv~~
~~fdivl~~
~~ddivl~~
~~ffma~~
~~ffmal~~
~~dfmal~~
~~fsqrt~~
~~fsqrtl~~
~~dsqrtl~~
~~canonicalize~~
~~canonicalizef~~
~~canonicalizel~~
The feature test macro __STDC_MATH_VERSION__ expands to the token *yyyymm*L.

3 The types

```
float_t
double_t
```

are floating types at least as wide as **float** and **double**, respectively, and such that **double_t** is at least as wide as **float_t**. If **FLT_EVAL_METHOD** equals 0, **float_t** and **double_t** are **float** and **double**, respectively; if **FLT_EVAL_METHOD** equals 1, they are both **double**; if **FLT_EVAL_METHOD** equals 2, they are both **long double**; and for other values of **FLT_EVAL_METHOD**, they are otherwise implementation-defined.[235)]

4 The macro

```
HUGE_VAL
```

[235)] The types **float_t** and **double_t** are intended to be the implementation's most efficient types at least as wide as **float** and **double**, respectively. For **FLT_EVAL_METHOD** equal 0, 1, or 2, the type **float_t** is the narrowest type used by the implementation to evaluate floating expressions.

expands to a positive `double` constant expression, not necessarily representable as a `float`. The macros

```
HUGE_VALF
HUGE_VALL
```

are respectively `float` and `long double` analogs of `HUGE_VAL`.[236]

5 The macro

```
INFINITY
```

expands to a constant expression of type `float` representing positive or unsigned infinity, if available; else to a positive constant of type `float` that overflows at translation time.[237]

6 The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs for the `float` type. It expands to a constant expression of type `float` representing a quiet NaN.

7 The *signaling NaN macros*

```
SNANF
SNAN
SNANL
```

each is defined if and only if the respective type contains signaling NaNs (5.2.4.2.2). They expand to a constant expression of the respective type representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.

8 The *number classification macros*

```
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

represent the mutually exclusive kinds of floating-point values. They expand to integer constant expressions with distinct values. Additional implementation-defined floating-point classifications, with macro definitions beginning with `FP_` and an uppercase letter, may also be specified by the implementation.

9 The *math rounding direction macros*

```
FP_INT_UPWARD
FP_INT_DOWNWARD
FP_INT_TOWARDZERO
FP_INT_TONEARESTFROMZERO
FP_INT_TONEAREST
```

represent the rounding directions of the functions `ceil`, `floor`, `trunc`, `round`, and `roundeven`, respectively, that convert to integral values in floating-point formats. They expand to integer constant expressions with distinct values suitable for use as the second argument to the ~~`fromfp`, `ufromfp`, `fromfpx`, and `ufromfpx`~~ `toint`, `touint`, `tointx`, and `touintx` functions.

[236] `HUGE_VAL`, `HUGE_VALF`, and `HUGE_VALL` can be positive infinities in an implementation that supports infinities.

[237] In this case, using `INFINITY` will violate the constraint in 6.4.4 and thus require a diagnostic.

```
        intmax_t toint(double x, int round, unsigned int width);
        intmax_t tointf(float x, int round, unsigned int width);
        intmax_t tointl(long double x, int round, unsigned int width);
        uintmax_t touint(double x, int round, unsigned int width);
        uintmax_t touintf(float x, int round, unsigned int width);
        uintmax_t touintl(long double x, int round, unsigned int width);
```

Description

2 The ~~fromfp and ufromfp~~ toint and touint functions round x, using the math rounding direction indicated by round, to a signed or unsigned integer, respectively, of width bits, and return the result value in the integer type designated by **intmax_t** or **uintmax_t**, respectively. If the value of the round argument is not equal to the value of a math rounding direction macro, the direction of rounding is unspecified. If the value of width exceeds the width of the function type, the rounding is to the full width of the function type. The ~~fromfp and ufromfp~~ toint and touint functions do not raise the “inexact” floating-point exception. If x is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type[248] of the specified width, or if width is zero, the functions return an unspecified value and a domain error occurs.

Returns

3 The ~~fromfp and ufromfp~~ toint and touint functions return the rounded integer value.

4 **EXAMPLE** Upward rounding of **double** x to type **int**, without raising the “inexact” floating-point exception, is achieved by

```
~~        (int)fromfp(x, FP_INT_UPWARD, INT_WIDTH)~~
        (int)toint(x, FP_INT_UPWARD, INT_WIDTH)
```

7.12.9.11 The tointx and touintx functions

Synopsis

1

```
~~        #define __STDC_WANT_IEC_60559_BFP_EXT__~~
        #include <stdint.h>
        #include <math.h>
~~        intmax_t fromfpx(double x, int round, unsigned int width);~~
~~        intmax_t fromfpxf(float x, int round, unsigned int width);~~
~~        intmax_t fromfpxl(long double x, int round, unsigned int width);~~
~~        uintmax_t ufromfpx(double x, int round, unsigned int width);~~
~~        uintmax_t ufromfpxf(float x, int round, unsigned int width);~~
~~        uintmax_t ufromfpxl(long double x, int round, unsigned int width);~~
        intmax_t tointx(double x, int round, unsigned int width);
        intmax_t tointxf(float x, int round, unsigned int width);
        intmax_t tointxl(long double x, int round, unsigned int width);
        uintmax_t touintx(double x, int round, unsigned int width);
        uintmax_t touintxf(float x, int round, unsigned int width);
        uintmax_t touintxl(long double x, int round, unsigned int width);
```

Description

2 The ~~fromfpx and ufromfpx~~ tointx and touintx functions differ from the ~~fromfp and ufromfp~~ toint and touint functions, respectively, only in that the ~~fromfpx and ufromfpx~~ tointx and touintx functions raise the “inexact” floating-point exception if a rounded result not exceeding the specified width differs in value from the argument x.

Returns

3 The ~~fromfpx and ufromfpx~~ tointx and touintx functions return the rounded integer value.

4 **NOTE** Conversions to integer types that are not required to raise the inexact exception can be done simply by rounding to integral value in floating type and then converting to the target integer type. For example, the conversion of **long double** x to **uint64_t**, using upward rounding, is done by

[248] For signed types, 6.2.6.2 permits three representations, which differ in whether a value of $-(2^M)$, where M is the number of value bits, can be represented.

```
      (uint64_t)ceill(x)
```

7.12.10 Remainder functions

7.12.10.1 The fmod functions

Synopsis

1

```
      #include <math.h>
      double fmod(double x, double y);
      float fmodf(float x, float y);
      long double fmodl(long double x, long double y);
```

Description

2 The **fmod** functions compute the floating-point remainder of x/y.

Returns

3 The **fmod** functions return the value x $-$ ny, for some integer n such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y. If y is zero, whether a domain error occurs or the **fmod** functions return zero is implementation-defined.

7.12.10.2 The remainder functions

Synopsis

1

```
      #include <math.h>
      double remainder(double x, double y);
      float remainderf(float x, float y);
      long double remainderl(long double x, long double y);
```

Description

2 The **remainder** functions compute the remainder x REM y required by IEC 60559.[249)]

Returns

3 The **remainder** functions return x REM y. If y is zero, whether a domain error occurs or the functions return zero is implementation defined.

7.12.10.3 The remquo functions

Synopsis

1

```
      #include <math.h>
      double remquo(double x, double y, int *quo);
      float remquof(float x, float y, int *quo);
      long double remquol(long double x, long double y, int *quo);
```

Description

2 The **remquo** functions compute the same remainder as the **remainder** functions. In the object pointed to by quo they store a value whose sign is the sign of x/y and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of x/y, where n is an implementation-defined integer greater than or equal to 3.

Returns

3 The **remquo** functions return x REM y. If y is zero, the value stored in the object pointed to by quo is unspecified and whether a domain error occurs or the functions return zero is implementation defined.

[249)] "When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = 1/2$, then n is even. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

7.20 Integer types `<stdint.h>`

1 The header `<stdint.h>` declares sets of integer types having specified widths, and defines corresponding sets of macros.[275] It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.

2 Types are defined in the following categories:

- — integer types having certain exact widths;
- — integer types having at least certain specified widths;
- — fastest integer types having at least certain specified widths;
- — integer types wide enough to hold pointers to objects;
- — integer types having greatest width.

(Some of these types may denote the same type.)

3 Corresponding macros specify limits of the declared types and construct suitable constants.

4 For each type described herein that the implementation provides,[276] `<stdint.h>` shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, `<stdint.h>` shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as "required", but need not provide any of the others (described as "optional").

5 ~~The following identifiers are defined only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where is first included: **INT*N*_WIDTH**~~
~~**UINT*N*_WIDTH**~~
~~***N*_WIDTH**~~
~~***N*_WIDTH**~~
~~***N*_WIDTH**~~
~~***N*_WIDTH**~~
~~**INTPTR_WIDTH**~~
~~**UINTPTR_WIDTH**~~
~~**INTMAX_WIDTH**~~
~~**UINTMAX_WIDTH**~~
~~**PTRDIFF_WIDTH**~~
~~**SIG_ATOMIC_WIDTH**~~
~~**SIZE_WIDTH**~~
~~**WCHAR_WIDTH**~~
~~**WINT_WIDTH**~~
The feature test macro **__STDC_STDINT_VERSION__** expands to the token *yyyymm*L.

7.20.1 Integer types

1 When typedef names differing only in the absence or presence of the initial `u` are defined, they shall denote corresponding signed and unsigned types as described in 6.2.5; an implementation providing one of these corresponding types shall also provide the other.

2 In the following descriptions, the symbol *N* represents an unsigned decimal integer with no leading zeros (e.g., 8 or 24, but not 04 or 048).

7.20.1.1 Exact-width integer types

1 The typedef name **int*N*_t** designates a signed integer type with width *N*, no padding bits, and a two's complement representation. Thus, **int8_t** denotes such a signed integer type with a width of exactly 8 bits.

[275] See "future library directions" (7.31.12).

[276] Some of these types might denote implementation-defined extended integer types.

7.22 General utilities <stdlib.h>

1 The header <stdlib.h> declares five types and several functions of general utility, and defines several macros.[307]

2 ~~The following identifiers are declared only if __STDC_WANT_IEC_60559_BFP_EXT__ is defined as a macro at the point in the source file where is first included: strfromd~~
~~strfromf~~
~~strfroml~~ The feature test macro __STDC_STDLIB_VERSION__ expands to the token *yyyymm*L.

3 The types declared are **size_t** and **wchar_t** (both described in 7.19),

```
div_t
```

which is a structure type that is the type of the value returned by the **div** function,

```
ldiv_t
```

which is a structure type that is the type of the value returned by the **ldiv** function, and

```
lldiv_t
```

which is a structure type that is the type of the value returned by the **lldiv** function.

4 The macros defined are **NULL** (described in 7.19);

```
EXIT_FAILURE
```

and

```
EXIT_SUCCESS
```

which expand to integer constant expressions that can be used as the argument to the **exit** function to return unsuccessful or successful termination status, respectively, to the host environment;

```
RAND_MAX
```

which expands to an integer constant expression that is the maximum value returned by the **rand** function; and

```
MB_CUR_MAX
```

which expands to a positive integer expression with type **size_t** that is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), which is never greater than **MB_LEN_MAX**.

7.22.1 Numeric conversion functions

1 The functions **atof**, **atoi**, **atol**, and **atoll** need not affect the value of the integer expression **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

7.22.1.1 The atof function

Synopsis

1
```
#include <stdlib.h>
double atof(const char *nptr);
```

[307] See "future library directions" (7.31.14).

Description

2 The **atof** function converts the initial portion of the string pointed to by nptr to **double** representation. Except for the behavior on error, it is equivalent to

```
       strtod(nptr, (char **)NULL)
```

Returns

3 The **atof** function returns the converted value.

Forward references: the **strtod**, **strtof**, and **strtold** functions (7.22.1.4).

7.22.1.2 The atoi, atol, and atoll functions

Synopsis

1
```
       #include <stdlib.h>
       int atoi(const char *nptr);
       long int atol(const char *nptr);
       long long int atoll(const char *nptr);
```

Description

2 The **atoi**, **atol**, and **atoll** functions convert the initial portion of the string pointed to by nptr to **int**, **long int**, and **long long int** representation, respectively. Except for the behavior on error, they are equivalent to

```
       atoi:  (int)strtol(nptr, (char **)NULL, 10)
       atol:  strtol(nptr, (char **)NULL, 10)
       atoll: strtoll(nptr, (char **)NULL, 10)
```

Returns

3 The **atoi**, **atol**, and **atoll** functions return the converted value.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.5).

7.22.1.3 The tostrd, tostrf, and tostrl functions

Synopsis

1
```
~~#define __STDC_WANT_IEC_60559_BFP_EXT__~~
#include <stdlib.h>
~~int strfromd(char *restrict s, size_t n, const char *restrict format, double fp);~~
~~int strfromf(char *restrict s, size_t n, const char *restrict format, float fp);~~
~~int strfroml(char *restrict s, size_t n, const char *restrict format, long double fp);~~
int tostrd(char *restrict s, size_t n, const char *restrict format, double fp);
int tostrf(char *restrict s, size_t n, const char *restrict format, float fp);
int tostrl(char *restrict s, size_t n, const char *restrict format, long double fp);
```

Description

2 The ~~**strfromd**, **strfromf**, and **strfroml**~~ **tostrd**, **tostrf**, and **tostrl** functions are equivalent to **snprintf**(s, n, format, fp) (7.21.6.5), except that the format string shall only contain the character %, an optional precision that does not contain an asterisk *, and one of the conversion specifiers a, A, e, E, f, F, g, or G, which applies to the type (**double**, **float**, or **long double**) indicated by the function suffix (rather than by a length modifier).

Returns

The ~~**strfromd**, **strfromf**, and **strfroml**~~ **tostrd**, **tostrf**, and **tostrl** functions return the number of characters that would have been written had n been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is less than n.

7.25 Type-generic math <tgmath.h>

1 The header <tgmath.h> includes the headers <math.h> and <complex.h> and defines several type-generic macros.

2 ~~The following identifiers are defined as type-generic macros only if __STDC_WANT_IEC_60559_BFP_EXT__ is defined as a macro at the point in the source file where is first included: roundevenllogbfmaxmagfminmagnextupnextdownfromfpufromfpfromfpxufromfpxfadddaddfsubdsubfmuldmul~~ feature test macro __STDC_TGMATH_VERSION__ expands to the token *yyyymm*L.

3 Of the <math.h> and <complex.h> functions without an **f** (**float**) or **l** (**long double**) suffix, several have one or more parameters whose corresponding real type is **double**. For each such function, except the functions that round result to narrower type (7.12.14) (which are covered below) and **modf**, there is a corresponding *type-generic macro*.[328] The parameters whose corresponding real type is **double** in the function synopsis are *generic parameters*. Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.[329]

4 Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

- — First, if any argument for generic parameters has type **long double**, the type determined is **long double**.
- — Otherwise, if any argument for generic parameters has type **double** or is of integer type, the type determined is **double**.
- — Otherwise, the type determined is **float**.

5 For each unsuffixed function in <math.h> for which there is a function in <complex.h> with the same name except for a **c** prefix, the corresponding type-generic macro (for both functions) has the same name as the function in <math.h>. The corresponding type-generic macro for **fabs** and **cabs** is **fabs**.

<math.h> function	<complex.h> function	type-generic macro
acos	**cacos**	**acos**
asin	**casin**	**asin**
atan	**catan**	**atan**
acosh	**cacosh**	**acosh**
asinh	**casinh**	**asinh**
atanh	**catanh**	**atanh**
cos	**ccos**	**cos**
sin	**csin**	**sin**
tan	**ctan**	**tan**
cosh	**ccosh**	**cosh**
sinh	**csinh**	**sinh**
tanh	**ctanh**	**tanh**
exp	**cexp**	**exp**
log	**clog**	**log**
pow	**cpow**	**pow**
sqrt	**csqrt**	**sqrt**
fabs	**cabs**	**fabs**

If at least one argument for a generic parameter is complex, then use of the macro invokes a complex function; otherwise, use of the macro invokes a real function.

[328] Like other function-like macros in standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

[329] If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.

6 For each unsuffixed function in <math.h> without a c-prefixed counterpart in <complex.h> (except functions that round result to narrower type, **modf**, and **canonicalize**), the corresponding type-generic macro has the same name as the function. These type-generic macros are:

atan2
cbrt
ceil
copysign
erf
erfc
exp2
expm1
fdim
floor
fma
fmax
fmaxmag
fmin
fminmag
fmod
frexp
~~**fromfp**~~**toint**
~~**fromfpx**~~**tointx**
hypot
ilogb
ldexp
lgamma
llogb
llrint
llround
log10
log1p
log2
logb
lrint
lround
nearbyint
nextafter
nextdown
nexttoward
nextup
remainder
remquo
rint
round
roundeven
scalbn
scalbln
tgamma
trunc
~~**ufromfp**~~**touint**
~~**ufromfpx**~~**touintx**

If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro is undefined.

7 For each unsuffixed function in <complex.h> that is not a c-prefixed counterpart to a function in <math.h>, the corresponding type-generic macro has the same name as the function. These type-generic macros are:

carg
cimag
conj
cproj
creal

Use of the macro with any real or complex argument invokes a complex function.

8 The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any l suffix[330) from the function names. Thus, the macros are:

fadd
dadd
fsub
dsub
fmul
dmul
fdiv
ddiv
ffma
dfma
fsqrt
dsqrt

All arguments shall be real. If any argument has type **long double**, or if the macro prefix is **d**, the function invoked has the name of the macro with an **l** suffix. Otherwise, the function invoked has the name of the macro (with no suffix).

9 A type-generic macro corresponding to a function indicated in the table in 7.6.2 is affected by constant rounding modes (7.6.3).

10 **NOTE** The type-generic macro definition in the example in 6.5.1.1 does not conform to this specification. A conforming macro could be implemented as follows:

```
      #define cbrt(X) _Generic((X),         \
            long double: cbrtl(X),         \
            default: _Roundwise_cbrt(X), \
            float: cbrtf(X)               \
            )
```

where _Roundwise_cbrt() is equivalent to **cbrt**() invoked without macro-replacement suppression.

[330) There are no functions with these macro names and the **f** suffix.

7.31 Future library directions

1 The following names are grouped under individual headers for convenience. All external names described below are reserved no matter what headers are included by the program.

7.31.1 Complex arithmetic `<complex.h>`

1 The function names

cerf
cerfc
cexp2
cexpm1
clog10
clog1p
clog2
clgamma
ctgamma

and the same names suffixed with **f** or **l** may be added to the declarations in the `<complex.h>` header.

7.31.2 Character handling `<ctype.h>`

1 Function names that begin with either **is** or **to**, and a lowercase letter may be added to the declarations in the `<ctype.h>` header.

7.31.3 Errors `<errno.h>`

1 Macros that begin with **E** and a digit or **E** and an uppercase letter may be added to the macros defined in the `<errno.h>` header.

7.31.4 Floating-point environment `<fenv.h>`

1 Macros that begin with **FE_** and an uppercase letter may be added to the macros defined in the `<fenv.h>` header.

7.31.5 Format conversion of integer types `<inttypes.h>`

1 Macros that begin with either **PRI** or **SCN**, and either a lowercase letter or **X** may be added to the macros defined in the `<inttypes.h>` header.

2 Function names that begin with **str**, or **wcs** and a lowercase letter may be added to the declarations in the `<inttypes.h>` header.

7.31.6 Localization `<locale.h>`

1 Macros that begin with **LC_** and an uppercase letter may be added to the macros defined in the `<locale.h>` header.

7.31.7 Mathematics `<math.h>`

1 Function names that begin with either **is** or **to**, and a lowercase letter may be added to the declarations in the `<math.h>` header.

2 Macros that begin with **DBL_**, **FLT_**, **FP_**, or **LDBL_** and an uppercase letter may be added to the macros defined in the `<math.h>` header.

7.31.8 Signal handling `<signal.h>`

1 Macros that begin with either **SIG** and an uppercase letter or **SIG_** and an uppercase letter may be added to the macros defined in the `<signal.h>` header.

7.31.9 Atomics `<stdatomic.h>`

1 Macros that begin with **ATOMIC_** and an uppercase letter may be added to the macros defined in the `<stdatomic.h>` header. Typedef names that begin with either **atomic_** or **memory_**, and a lowercase letter may be added to the declarations in the `<stdatomic.h>` header. Enumeration constants that begin with **memory_order_** and a lowercase letter may be added to the definition of the **memory_order** type in the `<stdatomic.h>` header. Function names that begin with **atomic_** and a lowercase letter may be added to the declarations in the `<stdatomic.h>` header.

2 The macro **ATOMIC_VAR_INIT** is an obsolescent feature.

7.31.10 Boolean type and values <stdbool.h>

1 The ability to undefine and perhaps then redefine the macros **bool**, **true**, and **false** is an obsolescent feature.

7.31.11 Integer types <stdint.h>

1 Typedef names beginning with **int** or **uint** and ending with **_t** may be added to the types defined in the <stdint.h> header. Macro names beginning with **INT** or **UINT** and ending with **_MAX**, **_MIN**, **_WIDTH**, or **_C** may be added to the macros defined in the <stdint.h> header.

7.31.12 Input/output <stdio.h>

1 Lowercase letters may be added to the conversion specifiers and length modifiers in **fprintf** and **fscanf**. Other characters may be used in extensions.

2 The use of **ungetc** on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

7.31.13 General utilities <stdlib.h>

1 Function names that begin with **str** or **wcs** and a lowercase letter may be added to the declarations in the <stdlib.h> header.

2 Invoking **realloc** with a size argument equal to zero is an obsolescent feature.

7.31.14 String handling <string.h>

1 Function names that begin with **str**, **mem**, or **wcs** and a lowercase letter may be added to the declarations in the <string.h> header.

7.31.15 Date and time <time.h>

Macros beginning with **TIME_** and an uppercase letter may be added to the macros in the <time.h> header.

7.31.16 Threads <threads.h>

1 Function names, type names, and enumeration constants that begin with either **cnd_**, **mtx_**, **thrd_**, or **tss_**, and a lowercase letter may be added to the declarations in the <threads.h> header.

7.31.17 Extended multibyte and wide character utilities <wchar.h>

1 Function names that begin with **wcs** and a lowercase letter may be added to the declarations in the <wchar.h> header.

2 Lowercase letters may be added to the conversion specifiers and length modifiers in **fwprintf** and **fwscanf**. Other characters may be used in extensions.

7.31.18 Wide character classification and mapping utilities <wctype.h>

1 Function names that begin with **is** or **to** and a lowercase letter may be added to the declarations in the <wctype.h> header.

```
rsize_t

errno_t memcpy_s(void *restrict s1, rsize_t s1max, const void *restrict s2, rsize_t n);
errno_t memmove_s(void *s1, rsize_t s1max, const void *s2, rsize_t n);
errno_t strcpy_s(char *restrict s1, rsize_t s1max, const char *restrict s2);
errno_t strncpy_s(
      char *restrict s1, rsize_t s1max, const char *restrict s2, rsize_t n);
errno_t strcat_s(char *restrict s1, rsize_t s1max, const char *restrict s2);
errno_t strncat_s(
      char *restrict s1, rsize_t s1max, const char *restrict s2, rsize_t n);
char *strtok_s(
      char *restrict s1, rsize_t *restrict s1max,
      const char *restrict s2, char **restrict ptr);
errno_t memset_s(void *s, rsize_t smax, int c, rsize_t n)
errno_t strerror_s(char *s, rsize_t maxsize, errno_t errnum);
size_t strerrorlen_s(errno_t errnum);
size_t strnlen_s(const char *s, size_t maxsize);
```

B.24 Type-generic math <tgmath.h>

acos
asin
atan
acosh
asinh
atanh
cos
sin
tan
cosh
sinh
tanh
exp
log
pow
sqrt
fabs
atan2
cbrt
ceil
copysign
erf
erfc
exp2
expm1
fdim
floor
fma
fmax
fmaxmag
fmin
fminmag
fmod
frexp
~~fromfp~~toint
~~fromfpx~~tointx
hypot
ilogb
ldexp
lgamma
llogb
llrint
llround
log10
log1p
log2
logb
lrint
lround
nearbyint
nextafter
nextdown
nexttoward
nextup
remainder
remquo
rint
round
roundeven
scalbn
scalbln
tgamma
trunc
~~ufromfp~~touint
~~ufromfpx~~touintx
carg
cimag
conj
cproj
creal
fadd
dadd
fsub
dsub
fmul
dmul
fdiv
ddiv
ffma
dfma
fsqrt
dsqrt

```
      __STDC_WANT_IEC_60559_BFP_EXT__

        totalorder
        totalordermag
```

B.25 Threads <threads.h>

__STDC_NO_THREADS__
thread_local
ONCE_FLAG_INIT
TSS_DTOR_ITERATIONS
cnd_t
thrd_t
tss_t
mtx_t
tss_dtor_t
thrd_start_t
once_flag
mtx_plain
mtx_recursive
mtx_timed
thrd_timedout
thrd_success
thrd_busy
thrd_error
thrd_nomem

Annex F
(normative)
IEC 60559 floating-point arithmetic

F.1 Introduction

1 This annex specifies C language support for the *IEC 60559* floating-point standard. The *IEC 60559 floating-point standard* is specifically *Floating-point arithmetic* (ISO/IEC/IEEE 60559:2011), also designated as *IEEE Standard for Floating-Point Arithmetic* (IEEE 754–2008). The IEC 60559 floating-point standard supersedes the IEC 60559:1989 *binary arithmetic standard*, also designated as *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754–1985). IEC 60559 generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc.

2 The IEC 60559 floating-point standard specifies decimal, as well as binary, floating-point arithmetic. It supersedes *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854–1987) which generalized the *binary arithmetic standard* (IEEE 754-1985) to remove dependencies on radix and word length.

3 An implementation that defines **__STDC_IEC_60559_BFP__** to *yyyymm*L shall conform to the specifications in this annex and shall also define **__STDC_IEC_559__** to 1.[373] Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

4 This annex amends some standard headers with declarations or definitions of identifiers contingent on whether certain macros whose names begin with **__STDC_WANT_IEC_60559_** and end with **_EXT__** are defined (by the user) at the point in the code where the header is first included. Within a preprocessing translation unit, the same set of such macros shall be defined for the first inclusion of all such headers.

F.2 Types

1 The C floating types match the IEC 60559 formats as follows:

- The **float** type matches the IEC 60559 binary32 format.
- The **double** type matches the IEC 60559 binary64 format.
- The **long double** type matches the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format,[374] else a non-IEC 60559 extended format, else the IEC 60559 binary64 format.

Any non-IEC 60559 extended format used for the **long double** type shall have more precision than IEC 60559 binary64 and at least the range of IEC 60559 binary64.[375] The value of **FLT_ROUNDS** applies to all IEC 60559 types supported by the implementation, but need not apply to non-IEC 60559 types.

Recommended practice

2 The **long double** type should match the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format.

F.2.1 Infinities and NaNs

1 Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values (5.2.4.2.2).

2 The **NAN** and **INFINITY** macros and the nan functions in `<math.h>` provide designations for IEC 60559 quiet NaNs and infinities. The **SNANF**, **SNAN**, and **SNANL** macros in `<math.h>` provide designations for IEC 60559 signaling NaNs.

[373] Implementations that do not define either of **__STDC_IEC_60559_BFP__** and **__STDC_IEC_559__** are not required to conform to these specifications. New code should not use the obsolescent macro **__STDC_IEC_559__** to test for conformance to this annex.

[374] IEC 60559 binary64-extended formats include the common 80-bit IEC 60559 format.

[375] A non-IEC 60559 **long double** type is required to provide infinity and NaNs, as its values include all **double** values.

3 This annex does not require the full support for signaling NaNs specified in IEC 60559. This annex uses the term NaN, unless explicitly qualified, to denote quiet NaNs. Where specification of signaling NaNs is not provided, the behavior of signaling NaNs is implementation-defined (either treated as an IEC 60559 quiet NaN or treated as an IEC 60559 signaling NaN).[376)]

4 Any operator or `<math.h>` function that raises an "invalid" floating-point exception, if delivering a floating type result, shall return a quiet NaN.

5 In order to support signaling NaNs as specified in IEC 60559, an implementation should adhere to the following recommended practice.

Recommended practice

6 Any floating-point operator or `<math.h>` function or macro with a signaling NaN input, unless explicitly specified otherwise, raises an "invalid" floating-point exception.

7 **NOTE** Some functions do not propagate quiet NaN arguments. For example, `hypot(x, y)` returns infinity if `x` or `y` is infinite and the other is a quiet NaN. The recommended practice in this subclause specifies that such functions (and others) raise the "invalid" floating-point exception if an argument is a signaling NaN, which also implies they return a quiet `NaN` in these cases.

8 The `<fenv.h>` header defines the macro **`FE_SNANS_ALWAYS_SIGNAL`** if and only if the implementation follows the recommended practice in this subclause. If defined, **`FE_SNANS_ALWAYS_SIGNAL`** expands to the integer constant `1`.

F.3 Operations

1 C operators, functions, and function-like macros provide the operations required by IEC 60559 as shown in the following table. Specifications for the C facilities are provided in the listed clauses. The C specifications are intended to match IEC 60559, unless stated otherwise.

Operation binding

IEC 60559 operation	C operation	Clause
roundToIntegralTiesToEven	`roundeven`	7.12.9.8, F.10.6.8
roundToIntegralTiesAway	`round`	7.12.9.6, F.10.6.6
roundToIntegralTowardZero	`trunc`	7.12.9.9, F.10.6.9
roundToIntegralTowardPositive	`ceil`	7.12.9.1, F.10.6.1
roundToIntegralTowardNegative	`floor`	7.12.9.2, F.10.6.2
roundToIntegralExact	`rint`	7.12.9.4, F.10.6.4
nextUp	`nextup`	7.12.11.5, F.10.8.5
nextDown	`nextdown`	7.12.11.6, F.10.8.6
remainder	`remainder, remquo`	7.12.10.2, F.10.7.2, 7.12.10.3, F.10.7.3
minNum	`fmin`	7.12.12.3, F.10.9.3
maxNum	`fmax`	7.12.12.2, F.10.9.2
minNumMag	`fminmag`	7.12.12.5, F.10.9.5
maxNumMag	`fmaxmag`	7.12.12.4, F.10.9.4
scaleB	`scalbn, scalbln`	7.12.6.14, F.10.3.14
logB	`logb`, `ilogb`, `llogb`	7.12.6.12, F.10.3.12, 7.12.6.5, F.10.3.5, 7.12.6.7, F.10.3.7
addition	`+`, `fadd`, `faddl`, `daddl`	6.5.6, 7.12.14.1, F.10.11
subtraction	`-`, `fsub`, `fsubl`, `dsubl`	6.5.6, 7.12.14.2, F.10.11
multiplication	`*`, `fmul`, `fmull`, `dmull`	6.5.5, 7.12.14.3, F.10.11

[376)] Since NaNs created by IEC 60559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

division	/, fdiv, fdivl, ddivl	6.5.5, 7.12.14.4, F.10.11
squareRoot	sqrt, fsqrt, fsqrtl, dsqrtl	7.12.7.5, F.10.4.5, 7.12.14.6, F.10.11
fusedMultiplyAdd	fma, ffma, ffmal, dfmal	7.12.13.1, F.10.10.1, 7.12.14.5, F.10.11
convertFromInt	cast and implicit conversion	6.3.1.4, 6.5.4
convertToIntegerTiesToEven convertToIntegerTowardZero convertToIntegerTowardPositive convertToIntegerTowardNegative	~~fromfp, ufromfp~~ toint, touint	??, ??
convertToIntegerTiesToAway	~~fromfp, ufromfp~~ toint, touint , lround, llround	??, ??, 7.12.9.7, F.10.6.7
convertToIntegerExactTiesToEven convertToIntegerExactTowardZero convertToIntegerExactTowardPositive convertToIntegerExactTowardNegative convertToIntegerExactTiesToAway	~~fromfpx, ufromfpx~~ tointx, touintx	??, ??
convertFormat - different formats	cast and implicit conversions	6.3.1.5, 6.5.4
convertFormat - same format	canonicalize	7.12.11.7, F.10.8.7
convertFromDecimalCharacter	strtod, wcstod, scanf, wscanf, decimal floating constants	7.22.1.4, 7.29.4.1.1, 7.21.6.4, 7.29.2.12, F.5
convertToDecimalCharacter	printf, wprintf, ~~strfromd~~ tostrd	7.21.6.3, 7.29.2.11, ??, F.5
convertFromHexCharacter	strtod, wcstod, scanf, wscanf, hexadecimal floating constants	7.22.1.4, 7.29.4.1.1, 7.21.6.4, 7.29.2.12, F.5
convertToHexCharacter	printf, wprintf, ~~strfromd~~ tostrd	7.21.6.3, 7.29.2.11, ??, F.5
copy	memcpy, memmove	7.24.2.1, 7.24.2.2
negate	- (x)	6.5.3.3
abs	fabs	7.12.7.2, F.10.4.2
copySign	copysign	7.12.11.1, F.10.8.1
compareQuietEqual	==	6.5.9, F.9.3
compareQuietNotEqual	!=	6.5.9, F.9.3
compareSignalingEqual	iseqsig	7.12.15.7, F.10.14.1
compareSignalingGreater	>	6.5.8, F.9.3
compareSignalingGreaterEqual	>=	6.5.8, F.9.3
compareSignalingLess	<	6.5.8, F.9.3
compareSignalingLessEqual	<=	6.5.8, F.9.3
compareSignalingNotEqual	! iseqsig(x)	7.12.15.7, F.10.14.1
compareSignalingNotGreater	! (x > y)	6.5.8, F.9.3
compareSignalingLessUnordered	! (x >= y)	6.5.8, F.9.3
compareSignalingNotLess	! (x < y)	6.5.8, F.9.3
compareSignalingGreaterUnordered	! (x <= y)	6.5.8, F.9.3
compareQuietGreater	isgreater	7.12.15.1
compareQuietGreaterEqual	isgreaterequal	7.12.15.2
compareQuietLess	isless	7.12.15.3
compareQuietLessEqual	islessequal	7.12.15.4
compareQuietUnordered	isunordered	7.12.15.6
compareQuietNotGreater	! isgreater(x, y)	7.12.15.1
compareQuietLessUnordered	! isgreaterequal(x, y)	7.12.15.2
compareQuietNotLess	! isless(x, y)	7.12.15.3

compareQuietGreaterUnordered	! islessequal(x, y)	7.12.15.4
compareQuietOrdered	! isunordered(x, y)	7.12.15.6
class	fpclassify, signbit, issignaling	7.12.3.1, 7.12.3.7, 7.12.3.8
isSignMinus	signbit	7.12.3.7
isNormal	isnormal	7.12.3.6
isFinite	isfinite	7.12.3.3
isZero	iszero	7.12.3.10
isSubnormal	issubnormal	7.12.3.9
isInfinite	isinf	7.12.3.4
isNaN	isnan	7.12.3.5
isSignaling	issignaling	7.12.3.8
isCanonical	iscanonical	7.12.3.2
radix	FLT_RADIX	5.2.4.2.2
totalOrder	totalorder	F.10.12.1
totalOrderMag	totalordermag	F.10.12.2
lowerFlags	feclearexcept	7.6.3.1
raiseFlags	fesetexcept	7.6.3.4
testFlags	fetestexcept	7.6.3.7
testSavedFlags	fetestexceptflag	7.6.3.6
restoreFlags	fesetexceptflag	7.6.3.5
saveAllFlags	fegetexceptflag	7.6.3.2
getBinaryRoundingDirection	fegetround	7.6.4.2
setBinaryRoundingDirection	fesetround	7.6.4.4
saveModes	fegetmode	7.6.4.1
restoreModes	fesetmode	7.6.4.3
defaultModes	fesetmode(FE_DFL_MODE)	7.6.4.3, 7.6

2 The IEC 60559 requirement that certain of its operations be provided for operands of different formats (of the same radix) is satisfied by C's usual arithmetic conversions (6.3.1.8) and function-call argument conversions (6.5.2.2). For example, the following operations take **float** f and **double** d inputs and produce a **long double** result:

```
(long double)f * d
powl(f, d)
```

3 Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation[377] is implementation-defined, even if <fenv.h> defines the macro FE_SNANS_ALWAYS_SIGNAL (F.2.1). If the return expression of a **return** statement is evaluated to the floating-point format of the return type, it is implementation-defined whether a convertFormat operation is applied to the result of the return expression.

4 The unary - operator raises no floating-point exceptions, even if the operand is a signaling NaN.

5 The C classification macros fpclassify, iscanonical, isfinite, isinf, isnan, isnormal, issignaling, issubnormal, and iszero provide the IEC 60559 operations indicated in the table above provided their arguments are in the format of their semantic type. Then these macros raise no floating-point exceptions, even if an argument is a signaling NaN.

6 The C nearbyint functions (7.12.9.3, F.10.6.3) provide the nearbyinteger function recommended in the Appendix to (superseded) ANSI/IEEE 854.

7 The C nextafter (7.12.11.3, F.10.8.3) and nexttoward (7.12.11.4, F.10.8.4) functions provide the

[377] Where the source and destination formats are the same, convertFormat operations differ from copy operations in that convertFormat operations raise the "invalid" floating-point exception on signaling NaN inputs and do not propagate non-canonical encodings.

nextafter function recommended in the Appendix to (superseded) IEC 60559:1989 (but with a minor change to better handle signed zeros).

8 The C **getpayload**, **setpayload**, and **setpayloadsig** (F.10.13) functions provide program access to NaN payloads, defined in IEC 60559.

9 The macros (7.6) **FE_DOWNWARD**, **FE_TONEAREST**, **FE_TOWARDZERO**, and **FE_UPWARD**, which are used in conjunction with the **fegetround** and **fesetround** functions and the **FENV_ROUND** pragma, represent the IEC 60559 rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTowardZero, and roundTowardPositive, respectively.

10 The C **fegetenv** (7.6.5.1), **feholdexcept** (7.6.5.2), **fesetenv** (7.6.5.3) and **feupdateenv** (7.6.5.4) functions provide a facility to manage the dynamic floating-point environment, comprising the IEC 60559 status flags and dynamic control modes.

11 IEC 60559 requires operations with specified operand and result formats. Therefore, math functions that are bound to IEC 60559 operations (see table above) must remove any extra range and precision from arguments or results.

12 IEC 60559 requires operations that round their result to formats the same as and wider than the operands, in addition to the operations that round their result to narrower formats (see 7.12.14). Operators (+,- ,*, and /) whose evaluation formats are wider than the semantic type (5.2.4.2.2) might not support some of the IEEE 60559 operations, because getting a result in a given format might require a cast that could introduce an extra rounding error. The functions that round result to narrower type (7.12.14) provide the IEC 60559 operations that round result to same and wider (as well as narrower) formats, in those cases where built-in operators and casts do not. For example, **ddivl**(x, y) computes a correctly rounded **double** divide of **float** x by **float** y, regardless of the evaluation method.

F.4 Floating to integer conversion

1 If the integer type is **_Bool**, 6.3.1.2 applies and the conversion raises no floating-point exceptions if the floating-point value is not a signaling NaN. Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the "invalid" floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by 6.3.1.4. Conversion of an integral floating value that does not exceed the range of the integer type raises no floating-point exceptions; whether conversion of a non-integral floating value raises the "inexact" floating-point exception is unspecified.[378)]

F.5 Conversions between binary floating types and decimal character sequences

1 Conversion from the widest supported IEC 60559 format to decimal with **DECIMAL_DIG** digits and back is the identity function.[379)]

2 Conversions involving IEC 60559 formats follow all pertinent recommended practice. In particular, conversion between any supported IEC 60559 format and decimal with **DECIMAL_DIG** or fewer significant digits is correctly rounded (honoring the current rounding mode), which assures that conversion from the widest supported IEC 60559 format to decimal with **DECIMAL_DIG** digits and back is the identity function.

3 The <float.h> header defines the macro

```
CR_DECIMAL_DIG
```

if and only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source

[378)] IEC 60559 recommends that implicit floating-to-integer conversions raise the "inexact" floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the "inexact" floating- point exception. See ~~fromfp~~ **toint** , ~~ufromfp~~ **touint** , ~~fromfpx~~ **tointx** , ~~ufromfpx~~ **touintx** , **rint**, **lrint**, **llrint**, and **nearbyint** in <math.h>.

[379)] If the minimum-width IEC 60559 binary64-extended format (64 bits of precision) is supported, **DECIMAL_DIG** is at least 21. If IEC 60559 binary64 (53 bits of precision) is the widest IEC 60559 format supported, then **DECIMAL_DIG** is at least 17. (By contrast, **LDBL_DIG** and **DBL_DIG** are 18 and 15, respectively, for these formats.)

file where <float.h> is first included. If defined, CR_DECIMAL_DIG expands to an integral constant expression suitable for use in #if preprocessing directives whose value is a number such that conversions between all supported types with IEC 60559 binary formats and character sequences with at most CR_DECIMAL_DIG significant decimal digits are correctly rounded. The value of CR_DECIMAL_DIG shall be at least DECIMAL_DIG + 3. If the implementation correctly rounds for all numbers of significant decimal digits, then CR_DECIMAL_DIG shall have the value of the macro UINTMAX_MAX.

4 Conversions of types with IEC 60559 binary formats to character sequences with more than CR_DECIMAL_DIG significant decimal digits shall correctly round to CR_DECIMAL_DIG significant digits and pad zeros on the right.

5 Conversions from character sequences with more than CR_DECIMAL_DIG significant decimal digits to types with IEC 60559 binary formats shall correctly round to an intermediate character sequence with CR_DECIMAL_DIG significant decimal digits, according to the applicable rounding direction, and correctly round the intermediate result (having CR_DECIMAL_DIG significant decimal digits) to the destination type. The "inexact" floating-point exception is raised (once) if either conversion is inexact.[380] (The second conversion may raise the "overflow" or "underflow" floating-point exception.)

6 Functions such as strtod that convert character sequences to floating types honor the rounding direction. Hence, if the rounding direction might be upward or downward, the implementation cannot convert a minus-signed sequence by negating the converted unsigned sequence.

7 The fprintf family of functions in <stdio.h> and the fwprintf family of functions in <wchar.h> should behave as if floating-point operands were passed through the canonicalize function of the same type.[381]

F.6 The return statement

If the return expression is evaluated in a floating-point format different from the return type, the expression is converted as if by assignment[382] to the return type of the function and the resulting value is returned to the caller.

F.7 Contracted expressions

1 A contracted expression is correctly rounded (once) and treats infinities, NaNs, signed zeros, subnormals, and the rounding directions in a manner consistent with the basic arithmetic operations covered by IEC 60559.

Recommended practice

2 A contracted expression should raise floating-point exceptions in a manner generally consistent with the basic arithmetic operations.

F.8 Floating-point environment

1 The floating-point environment defined in <fenv.h> includes the IEC 60559 floating-point exception status flags and directed-rounding control modes. It includes also IEC 60559 dynamic rounding precision and trap enablement modes, if the implementation supports them.[383]

F.8.1 Environment management

1 IEC 60559 requires that floating-point operations implicitly raise floating-point exception status flags, and that rounding control modes can be set explicitly to affect result values of floating-point operations. These changes to the floating-point state are treated as side effects which respect sequence points.[384]

[380] The intermediate conversion is exact only if all input digits after the first CR_DECIMAL_DIG digits are 0.

[381] This is a recommendation instead of a requirement so that implementations may choose to print signaling NaNs differently from quiet NaNs.

[382] Assignment removes any extra range and precision.

[383] This specification does not require dynamic rounding precision nor trap enablement modes.

[384] If the state for the FENV_ACCESS pragma is "off", the implementation is free to assume the dynamic floating-point control modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see

- `trunc`(± 0) returns ± 0.
- `trunc`($\pm\infty$) returns $\pm\infty$.

2 The returned value is exact and is independent of the current rounding direction mode.

F.10.6.10 The `toint` and `touint` functions

1 The ~~`fromfp` and `ufromfp`~~ `toint` and `touint` functions raise the "invalid" floating-point exception and return an unspecified value if the floating-point argument `x` is infinite or `NaN` or rounds to an integral value that is outside the range of any supported integer type of the specified width.

2 These functions do not raise the "inexact" floating-point exception.

F.10.6.11 The `tointx` and `touintx` functions

1 The ~~`fromfpx` and `ufromfpx`~~ `tointx` and `touintx` functions raise the "invalid" floating-point exception and return an unspecified value if the floating-point argument `x` is infinite or `NaN` or rounds to an integral value that is outside the range of any supported integer type of the specified width.

2 These functions raise the "inexact" floating-point exception if a valid result differs in value from the floating-point argument `x`.

F.10.7 Remainder functions

F.10.7.1 The `fmod` functions

1
- `fmod`($\pm 0, y$) returns ± 0 for y not zero.
- `fmod`(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).
- `fmod`($x, \pm\infty$) returns x for x not infinite.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

3 The **`double`** version of `fmod` behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double fmod(double x, double y)
{
      double result;
      result = remainder(fabs(x), (y = fabs(y)));
      if (signbit(result)) result += y;
      return copysign(result, x);
}
```

F.10.7.2 The `remainder` functions

1
- `remainder`($\pm 0, y$) returns ± 0 for y not zero.
- `remainder`(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).
- `remainder`($x, \pm\infty$) returns x for x *not infinite*.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

F.10.7.3 The `remquo` functions

1 The `remquo` functions follow the specifications for the `remainder` functions. They have no further specifications special to IEC 60559 implementations.

2 When subnormal results are supported, the returned value is exact and is independent of the current rounding direction mode.

J.5.11 Multiple external definitions

1 There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **extern**; if the definitions disagree, or more than one is initialized, the behavior is undefined (6.9.2).

J.5.12 Predefined macro names

1 Macro names that do not begin with an underscore, describing the translation and execution environments, are defined by the implementation before translation begins (6.10.8).

J.5.13 Floating-point status flags

1 If any floating-point status flags are set on normal termination after all calls to functions registered by the **atexit** function have been made (see 7.22.4.4), the implementation writes some diagnostics indicating the fact to the **stderr** stream, if it is still open,

J.5.14 Extra arguments for signal handlers

1 Handlers for specific signals are called with extra arguments in addition to the signal number (7.14.1.1).

J.5.15 Additional stream types and file-opening modes

1 Additional mappings from files to streams are supported (7.21.2).

2 Additional file-opening modes may be specified by characters appended to the mode argument of the **fopen** function (7.21.5.3).

J.5.16 Defined file position indicator

1 The file position indicator is decremented by each successful call to the **ungetc** or **ungetwc** function for a text stream, except if its value was zero before a call (7.21.7.10, 7.29.3.10).

J.5.17 Math error reporting

1 Functions declared in <complex.h> and <math.h> raise **SIGFPE** to report errors instead of, or in addition to, setting **errno** or raising floating-point exceptions (7.3, 7.12).

J.6 Reserved identifiers and keywords

1 A lot of identifier preprocessing tokens are used for specific purposes in regular clauses or appendices from translation phase 3 onwards. Using any of these for a purpose different from their description in this document, even if the use is in a context where they are normatively permitted, may have an impact on the portability of code and should thus be avoided.

J.6.1 Rule based identifiers

1 The following ~~29~~ 33 regular expressions characterize identifiers that are systematically reserved by some clause this document.

```
atomic_[a-z][a-zA-Z0-9_]*
ATOMIC_[A-Z][a-zA-Z0-9_]*
_[a-zA-Z_][a-zA-Z0-9_]*
cnd_[a-z][a-zA-Z0-9_]*
DBL_[A-Z][a-zA-Z0-9_]*
E[0-9A-Z][a-zA-Z0-9_]*
FE_[A-Z][a-zA-Z0-9_]*
FLT_[A-Z][a-zA-Z0-9_]*
FP_[A-Z][a-zA-Z0-9_]*
INT[a-zA-Z0-9_]*_C
INT[a-zA-Z0-9_]*_MAX
INT[a-zA-Z0-9_]*_MIN
int[a-zA-Z0-9_]*_t
INT[a-zA-Z0-9_]*_WIDTH
is[a-z][a-zA-Z0-9_]*
LC_[A-Z][a-zA-Z0-9_]*
mem[a-z][a-zA-Z0-9_]*
mtx_[a-z][a-zA-Z0-9_]*
LDBL_[A-Z][a-zA-Z0-9_]*
PRI[a-zX][a-zA-Z0-9_]*
SCN[a-zX][a-zA-Z0-9_]*
SIG[A-Z][a-zA-Z0-9_]*
SIG_[A-Z][a-zA-Z0-9_]*
str[a-z][a-zA-Z0-9_]*
thrd_[a-z][a-zA-Z0-9_]*
TIME_[A-Z][a-zA-Z0-9_]*
to[a-z][a-zA-Z0-9_]*
tss_[a-z][a-zA-Z0-9_]*
```

```
UINT[a-zA-Z0-9_]*_C
UINT[a-zA-Z0-9_]*_MAX
uint[a-zA-Z0-9_]*_t
UINT[a-zA-Z0-9_]*_WIDTH
wcs[a-z][a-zA-Z0-9_]*
```

2 The following ~~462~~ 554 identifiers or keywords match these patterns and have particular semantics provided by this document.

```
_Alignas
__alignas_is_defined
_Alignof
__alignof_is_defined
_Atomic
atomic_bool
ATOMIC_BOOL_LOCK_FREE
atomic_char
atomic_char16_t
ATOMIC_CHAR16_T_LOCK_FREE
atomic_char32_t
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_CHAR_LOCK_FREE
atomic_compare_exchange_strong
atomic_compare_exchange_strong_explicit
atomic_compare_exchange_weak
atomic_compare_exchange_weak_explicit
atomic_exchange
atomic_exchange_explicit
atomic_fetch_
atomic_fetch_add
atomic_fetch_add_explicit
atomic_fetch_and
atomic_fetch_and_explicit
atomic_fetch_or
atomic_fetch_or_explicit
atomic_fetch_sub
atomic_fetch_sub_explicit
atomic_fetch_xor
atomic_fetch_xor_explicit
atomic_flag
atomic_flag_clear
atomic_flag_clear_explicit
ATOMIC_FLAG_INIT
atomic_flag_test_and_set
atomic_flag_test_and_set_explicit
atomic_init
atomic_int
atomic_int_fast16_t
atomic_int_fast32_t
atomic_int_fast64_t
atomic_int_fast8_t
atomic_int_least16_t
atomic_int_least32_t
atomic_int_least64_t
atomic_int_least8_t
ATOMIC_INT_LOCK_FREE
atomic_intmax_t
atomic_intptr_t
atomic_is_lock_free
atomic_llong
ATOMIC_LLONG_LOCK_FREE
atomic_load
atomic_load_explicit
atomic_long
ATOMIC_LONG_LOCK_FREE
ATOMIC_POINTER_LOCK_FREE
atomic_ptrdiff_t
atomic_schar
atomic_short
ATOMIC_SHORT_LOCK_FREE
atomic_signal_fence
atomic_size_t
atomic_store
atomic_store_explicit
atomic_thread_fence
atomic_uchar
atomic_uint
atomic_uint_fast16_t
atomic_uint_fast32_t
atomic_uint_fast64_t
atomic_uint_fast8_t
atomic_uint_least16_t
atomic_uint_least32_t
atomic_uint_least64_t
atomic_uint_least8_t
atomic_uintmax_t
atomic_uintptr_t
atomic_ullong
atomic_ulong
atomic_ushort
ATOMIC_VAR_INIT
atomic_wchar_t
ATOMIC_WCHAR_T_LOCK_FREE
_Bool
__bool_true_false_are_defined
cnd_broadcast
cnd_destroy
cnd_init
cnd_signal
cnd_t
cnd_timedwait
cnd_wait
```

~~_Complex~~ DBL_DECIMAL_DIG

~~_Complex_I~~ DBL_DIG

~~__cplusplus~~ DBL_EPSILON

~~__DATE__~~ DBL_HAS_SUBNORM

DBL_MANT_DIG

DBL_MAX
DBL_MAX_10_EXP
DBL_MAX_EXP
DBL_MIN
DBL_MIN_10_EXP
DBL_MIN_EXP
DBL_TRUE_MIN
EDOM
EILSEQ
EOF
EOL
ERANGE
_Exit
EXIT_FAILURE
EXIT_SUCCESS
_EXT__
FE_ALL_EXCEPT
FE_DFL_ENV
FE_DFL_MODE
FE_DIVBYZERO
FE_DOWNWARD
FE_DYNAMIC
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_SNANS_ALWAYS_SIGNAL
FE_TONEAREST
FE_TOWARDZERO
FE_UNDERFLOW
FE_UPWARD
~~__FILE__~~ FLT_DECIMAL_DIG
~~__func__~~ FLT_DIG
~~_Generic~~ FLT_EPSILON
~~_Imaginary~~ FLT_EVAL_METHOD
~~_Imaginary_I~~ FLT_HAS_SUBNORM
FLT_MANT_DIG
FLT_MAX
FLT_MAX_10_EXP
FLT_MAX_EXP
FLT_MIN
FLT_MIN_10_EXP
FLT_MIN_EXP
FLT_RADIX
FLT_ROUNDS
FLT_TRUE_MIN
FP_CONTRACT
FP_FAST_DADDL
FP_FAST_DDIVL
FP_FAST_DFMAL
FP_FAST_DMULL
FP_FAST_DSQRTL
FP_FAST_DSUBL
FP_FAST_FADD
FP_FAST_FADDL
FP_FAST_FDIV
FP_FAST_FDIVL
FP_FAST_FFMA
FP_FAST_FFMAL
FP_FAST_FMA
FP_FAST_FMAF
FP_FAST_FMAL
FP_FAST_FMUL
FP_FAST_FMULL
FP_FAST_FSQRT
FP_FAST_FSQRTL
FP_FAST_FSUB
FP_FAST_FSUBL
FP_ILOGB0
FP_ILOGBNAN
FP_INFINITE
FP_INT_DOWNWARD
FP_INT_TONEAREST
FP_INT_TONEARESTFROMZERO
FP_INT_TOWARDZERO
FP_INT_UPWARD
FP_LLOGB0
FP_LLOGBNAN
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
INT16_C
INT16_MAX
INT16_MIN
int16_t
INT32_C
INT32_MAX
INT32_MIN
int32_t
INT64_C
INT64_MAX
INT64_MIN
int64_t
INT8_C
INT8_MAX
INT8_MIN
int8_t
int_fast16_t
int_fast32_t
int_fast64_t
int_fast8_t
int_least16_t
int_least32_t
int_least64_t
int_least8_t
INT_MAX
INTMAX_C
INTMAX_MAX
INTMAX_MIN
intmax_t
INTMAX_WIDTH
INT_MIN

INTPTR_MAX
INTPTR_MIN
intptr_t
INTPTR_WIDTH
INT_WIDTH
_IOFBF
_IOLBF
_IONBF
isalnum
isalpha
isblank
iscanonical
iscntrl
isdigit
iseqsig
isfinite
isgraph
isgreater
isgreaterequal
isinf
isless
islessequal
islessgreater
islower
isnan
isnormal
isprint
ispunct
issignaling
isspace
issubnormal
isunordered
isupper
iswalnum
iswalpha
iswblank
iswcntrl
iswctype
iswdigit
iswgraph
iswlower
iswprint
iswpunct
iswspace
iswupper
iswxdigit
isxdigit
iszero
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
~~__LINE__~~LDBL_DECIMAL_DIG
LDBL_DIG
LDBL_EPSILON
LDBL_HAS_SUBNORM
LDBL_MANT_DIG
LDBL_MAX
LDBL_MAX_10_EXP
LDBL_MAX_EXP
LDBL_MIN
LDBL_MIN_10_EXP
LDBL_MIN_EXP
LDBL_TRUE_MIN
memchr
memcmp
memcpy
memcpy_s
memmove
memmove_s
memory_order
memory_order_acq_rel
memory_order_acquire
memory_order_consume
memory_order_relaxed
memory_order_release
memory_order_seq_cst
memset
memset_s
mtx_destroy
mtx_init
mtx_lock
mtx_plain
mtx_recursive
mtx_t
mtx_timed
mtx_timedlock
mtx_trylock
mtx_unlock
_Noreturn
_Pragma
PRId32
PRId64
PRIdFAST32
PRIdFAST64
PRIdLEAST32
PRIdLEAST64
PRIdMAX
PRIdPTR
PRIi32
PRIi64
PRIiFAST32
PRIiFAST64
PRIiLEAST32
PRIiLEAST64
PRIiMAX
PRIiPTR
PRIo32
PRIo64
PRIoFAST32

PRIoFAST64
PRIoLEAST32
PRIoLEAST64
PRIoMAX
PRIoPTR
PRIu32
PRIu64
PRIuFAST32
PRIuFAST64
PRIuLEAST32
PRIuLEAST64
PRIuMAX
PRIuPTR
PRIX32
PRIX64
PRIXFAST32
PRIXFAST64
PRIXLEAST32
PRIXLEAST64
PRIXMAX
PRIXPTR
SCNdMAX
SCNdPTR
SCNiMAX
SCNiPTR
SCNoMAX
SCNoPTR
SCNuMAX
SCNuPTR
SCNxMAX
SCNxPTR
SIGABRT
SIG_ATOMIC_MAX
SIG_ATOMIC_MIN
SIG_ATOMIC_WIDTH
SIG_DFL
SIG_ERR
SIGFPE
SIG_IGN
SIGILL
SIGINT
SIGSEGV
SIGTERM
_Static_assert
__STDC__
__STDC_ANALYZABLE__
__STDC_HOSTED__
__STDC_IEC_559__
__STDC_IEC_559_COMPLEX__
__STDC_IEC_60559_BFP__
__STDC_IEC_60559_COMPLEX__
__STDC_ISO_10646__
__STDC_LIB_EXT1__
__STDC_MB_MIGHT_NEQ_WC__
__STDC_NO_ATOMICS__
__STDC_NO_COMPLEX__
__STDC_NO_THREADS__
__STDC_NO_VLA__
__STDC_UTF_16__
__STDC_UTF_32__
__STDC_VERSION__
__STDC_WANT_IEC_60559_
__STDC_WANT_IEC_60559_BFP_EXT__
__STDC_WANT_LIB_EXT1__
strcat
strcat_s
strchr
strcmp
strcoll
strcpy
strcpy_s
strcspn
strerror
strerrorlen_s
strerror_s
~~strfromd~~
~~strfromf~~
~~strfroml~~
strftime
strlen
strncat
strncat_s
strncmp
strncpy
strncpy_s
strnlen_s
strpbrk
strrchr
strspn
strstr
strtod
strtof
strtoimax
strtok
strtok_s
strtol
strtold
strtoll
strtoul
strtoull
strtoumax
struct
strxfrm
thrd_busy
thrd_create
thrd_current
thrd_detach
thrd_equal
thrd_error
thrd_exit
thrd_join
thrd_nomem

thrd_sleep
thrd_start_t
thrd_success
thrd_t
thrd_timedout
thrd_yield
_Thread_local
__TIME__
TIME_UTC
toint
tointf
tointl
tointx
tointxf
tointxl
tolower
tostrd
tostrf
tostrld
totalorder
totalorderf
totalorderl
totalordermag
totalordermagf
totalordermagl
touint
touintf
touintl
touintx
touintxf
touintxl
toupper
towctrans
towlower
towupper
tss_create
tss_delete
tss_dtor_t
tss_get
tss_set
tss_t
UINT16_C
UINT16_MAX
uint16_t
UINT32_C
UINT32_MAX
uint32_t
UINT64_C
UINT64_MAX
uint64_t
UINT8_C
UINT8_MAX
uint8_t
uint_fast16_t
uint_fast32_t
uint_fast64_t
uint_fast8_t
uint_least16_t
uint_least32_t
uint_least64_t
uint_least8_t
UINT_MAX
UINTMAX_C
UINTMAX_MAX
uintmax_t
UINTMAX_WIDTH
UINTPTR_MAX
uintptr_t
UINTPTR_WIDTH
UINT_WIDTH
__VA_ARGS__
wcscat
wcscat_s
wcschr
wcscmp
wcscoll
wcscpy
wcscpy_s
wcscspn
wcsftime
wcslen
wcsncat
wcsncat_s
wcsncmp
wcsncpy
wcsncpy_s
wcsnlen_s
wcspbrk
wcsrchr
wcsrtombs
wcsrtombs_s
wcsspn
wcsstr
wcstod
wcstof
wcstoimax
wcstok
wcstok_s
wcstol
wcstold
wcstoll
wcstombs
wcstombs_s
wcstoul
wcstoull
wcstoumax
wcsxfrm
~~_WIDTH~~ _Alignas
_Alignof
_Atomic
_Bool
_Complex

_Complex_I
_Exit
_EXT__
_Generic
_Imaginary
_Imaginary_I
_IOFBF
_IOLBF
_IONBF
_Noreturn
_Pragma
_Static_assert
_Thread_local
_WIDTH
__alignas_is_defined
__alignof_is_defined
__bool_true_false_are_defined
__cplusplus
__DATE__
__FILE__
__func__
__LINE__
__STDC_ANALYZABLE__
__STDC_FENV_VERSION__
__STDC_HOSTED__
__STDC_IEC_559_COMPLEX__
__STDC_IEC_559__
__STDC_IEC_60559_BFP__
__STDC_IEC_60559_COMPLEX__
__STDC_ISO_10646__
__STDC_LIB_EXT1__
__STDC_MATH_VERSION__
__STDC_MB_MIGHT_NEQ_WC__
__STDC_NO_ATOMICS__
__STDC_NO_COMPLEX__
__STDC_NO_THREADS__
__STDC_NO_VLA__
__STDC_STDINT_VERSION__
__STDC_STDLIB_VERSION__
__STDC_TGMATH_VERSION__
__STDC_UTF_16__
__STDC_UTF_32__
__STDC_VERSION__
__STDC_WANT_IEC_60559_
__STDC_WANT_IEC_60559_BFP_EXT__
__STDC_WANT_LIB_EXT1__
__STDC__
__TIME__
__VA_ARGS__

J.6.2 Particular identifiers or keywords

1 The following ~~808~~ 721 identifiers or keywords are not covered by the above and have particular semantics provided by this document.

abort
abort_handler_s
abs
acos
acosf
acosh
acoshf
acoshl
acosl
alignas
aligned_alloc
alignof
and
and_eq
asctime
asctime_s
asin
asinf
asinh
asinhf
asinhl
asinl
assert
atan
atan2
atan2f
atan2l
atanf
atanh
atanhf
atanhl
atanl
atexit
atof
atoi
atol
atoll
at_quick_exit
auto
bitand
bitor
bool
break
bsearch
bsearch_s
btowc
BUFSIZ
c16rtomb
c32rtomb
cabs
cabsf
cabsl
cacos
cacosf
cacosh
cacoshf
cacoshl
cacosl
calloc
call_once
canonicalize
canonicalizef
canonicalizel
carg
cargf
cargl
case
casin
casinf
casinh
casinhf
casinhl
casinl
catan
catanf

catanh
catanhf
catanhl
catanl
cbrt
cbrtf
cbrtl
ccos
ccosf
ccosh
ccoshf
ccoshl
ccosl
ceil
ceilf
ceill
cerf
cerfc
cexp
cexp2
cexpf
cexpl
cexpm1
char
char16_t
char32_t
CHAR_BIT
CHAR_MAX
CHAR_MIN
CHAR_WIDTH
cimag
cimagf
cimagl
clearerr
clgamma
clock
CLOCKS_PER_SEC
clock_t
clog
clog10
clog1p
clog2
clogf
clogl
CMPLX
CMPLXF
CMPLXL
compl
complex
conj
conjf
conjl
const
constraint_handler_t
continue
copysign
copysignf
copysignl
cos
cosf
cosh
coshf
coshl
cosl
cpow
cpowf
cpowl
cproj
cprojf
cprojl
CR_DECIMAL_DIG
creal
crealf
creall
csin
csinf
csinh
csinhf
csinhl
csinl
csqrt
csqrtf
csqrtl
ctan
ctanf
ctanh
ctanhf
ctanhl
ctanl
ctgamma
ctime
ctime_s
currency_symbol
CX_LIMITED_RANGE
dadd
~~daddl~~
~~DBL_DECIMAL_DIG~~
~~DBL_DIG~~
~~DBL_EPSILON~~
~~DBL_HAS_SUBNORM~~
~~DBL_MANT_DIG~~
~~DBL_MAX~~
~~DBL_MAX_10_EXP~~
~~DBL_MAX_EXP~~
~~DBL_MIN~~
~~DBL_MIN_10_EXP~~
~~DBL_MIN_EXP~~
~~DBL_TRUE_MIN~~ ddivl
ddiv
ddivl
DECIMAL_DIG
decimal_point
DEFAULT
define
defined
dfma
dfmal
difftime
div
div_t
dmul
dmull
do
double
double_t
dsqrt
dsqrtl
dsub
dsubl
elif
else
endif
enum
erf
erfc
erfcf
erfcl
erff
erfl
errno
errno_t
error
exit
exp
exp2
exp2f
exp2l
expf
expl
expm1
expm1f
expm1l
extern
fabs
fabsf
fabsl
fadd
faddl
false
fclose
fdim
fdimf
fdiml
fdiv
fdivl
feclearexcept
fegetenv
fegetexceptflag

fegetmode
fegetround
feholdexcept
femode_t
FENV_ACCESS
FENV_ROUND
fenv_t
feof
feraiseexcept
ferror
fesetenv
fesetexcept
fesetexceptflag
fesetmode
fesetround
fetestexcept
fetestexceptflag
feupdateenv
fexcept_t
fflush
ffma
ffmal
fgetc
fgetpos
fgets
fgetwc
fgetws
FILE
FILENAME_MAX
float
float_t
floor
floorf
floorl
~~FLT_DECIMAL_DIG~~
~~FLT_DIG~~
~~FLT_EPSILON~~
~~FLT_EVAL_METHOD~~
~~FLT_HAS_SUBNORM~~
~~FLT_MANT_DIG~~
~~FLT_MAX~~
~~FLT_MAX_10_EXP~~
~~FLT_MAX_EXP~~
~~FLT_MIN~~
~~FLT_MIN_10_EXP~~
~~FLT_MIN_EXP~~
~~FLT_RADIX~~
~~FLT_ROUNDS~~
~~FLT_TRUE_MIN~~
~~fma~~ floor
fmaf
fmal
fmax
fmaxf
fmaxl
fmaxmag
fmaxmagf
fmaxmagl
fmin
fminf
fminl
fminmag
fminmagf
fminmagl
fmod
fmodf
fmodl
fmul
fmull
fopen
FOPEN_MAX
fopen_s
for
fpclassify
~~FP_CONTRACT~~
~~FP_FAST_DADDL~~
~~FP_FAST_DDIVL~~
~~FP_FAST_DFMAL~~
~~FP_FAST_DMULL~~
~~FP_FAST_DSQRTL~~
~~FP_FAST_DSUBL~~
~~FP_FAST_FADD~~
~~FP_FAST_FADDL~~
~~FP_FAST_FDIV~~
~~FP_FAST_FDIVL~~
~~FP_FAST_FFMA~~
~~FP_FAST_FFMAL~~
~~FP_FAST_FMA~~
~~FP_FAST_FMAF~~
~~FP_FAST_FMAL~~
~~FP_FAST_FMUL~~
~~FP_FAST_FMULL~~
~~FP_FAST_FSQRT~~
~~FP_FAST_FSQRTL~~
~~FP_FAST_FSUB~~
~~FP_FAST_FSUBL~~
~~FP_ILOGB0~~
~~FP_ILOGBNAN~~
~~FP_INFINITE~~
~~FP_INT_DOWNWARD~~
~~FP_INT_TONEAREST~~
~~FP_INT_TONEARESTFROMZERO~~
~~FP_INT_TOWARDZERO~~
~~FP_INT_UPWARD~~
~~FP_LLOGB0~~
~~FP_LLOGBNAN~~
~~FP_NAN~~
~~FP_NORMAL~~
fpos_t
~~fprintf~~
fprintf_s
~~FP_SUBNORMAL~~ fprintf
fputc
fputs
fputwc
fputws
~~FP_ZERO~~
frac_digits
fread
free
freopen
freopen_s
frexp
frexpf
frexpl
~~fromfp~~
~~fromfpf~~
~~fromfpl~~
~~fromfpx~~
~~fromfpxf~~ frexp
~~fromfpxl~~ fscanf_s
fscanf
fscanf_s
fseek
fsetpos
fsqrt
fsqrtl
fsub
fsubl
ftell
fwide
fwprintf
fwprintf_s
fwrite
fwscanf
fwscanf_s
getc
getchar
getenv
getenv_s
getpayload
getpayloadf
getpayloadl
gets
gets_s
getwc
getwchar
gmtime
gmtime_s
goto
grouping
HUGE_VAL
HUGE_VALF
HUGE_VALL
hypot
hypotf
hypotl
I

if
ifdef
ifndef
ignore_handler_s
ilogb
ilogbf
ilogbl
imaginary
imaxabs
imaxdiv
imaxdiv_t
include
INFINITY
inline
int_curr_symbol
int_frac_digits
int_n_cs_precedes
int_n_sep_by_space
int_n_sign_posn
int_p_cs_precedes
int_p_sep_by_space
int_p_sign_posn
jmp_buf
kill_dependency
labs
lconv
~~LDBL_DECIMAL_DIG~~
~~LDBL_DIG~~
~~LDBL_EPSILON~~
~~LDBL_HAS_SUBNORM~~
~~LDBL_MANT_DIG~~
~~LDBL_MAX~~
~~LDBL_MAX_10_EXP~~
~~LDBL_MAX_EXP~~
~~LDBL_MIN~~
~~LDBL_MIN_10_EXP~~
~~LDBL_MIN_EXP~~
~~LDBL_TRUE_MIN~~
~~ldexp~~
ldexpf
ldexpl
ldiv
ldiv_t
lgamma
lgammaf
lgammal
line
llabs
lldiv
lldiv_t
llogb
llogbf
llogbl
LLONG_MAX
LLONG_MIN
LLONG_WIDTH
llrint
llrintf
llrintl
llround
llroundf
llroundl
localeconv
localtime
localtime_s
log
log10
log10f
log10l
log1p
log1pf
log1pl
log2
log2f
log2l
logb
logbf
logbl
logf
logl
long
longjmp
LONG_MAX
LONG_MIN
LONG_WIDTH
lrint
lrintf
lrintl
lround
lroundf
lroundl
L_tmpnam
L_tmpnam_s
main
malloc
MATH_ERREXCEPT
math_errhandling
MATH_ERRNO
max_align_t
MB_CUR_MAX
mblen
MB_LEN_MAX
mbrlen
mbrtoc16
mbrtoc32
mbrtowc
mbsinit
mbsrtowcs
mbsrtowcs_s
mbstate_t
mbstowcs
mbstowcs_s
mbtowc
mktime
modf
modff
modfl
mon_decimal_point
mon_grouping
mon_thousands_sep
nan
nanf
nanl
n_cs_precedes
NDEBUG
nearbyint
nearbyintf
nearbyintl
negative_sign
nextafter
nextafterf
nextafterl
nextdown
nextdownf
nextdownl
nexttoward
nexttowardf
nexttowardl
nextup
nextupf
nextupl
noreturn
not
not_eq
n_sep_by_space
n_sign_posn
NULL
OFF
offsetof
ON
once_flag
ONCE_FLAG_INIT
or
or_eq
p_cs_precedes
perror
positive_sign
pow
powf
powl
pragma
printf
printf_s
p_sep_by_space
p_sign_posn
PTRDIFF_MAX
PTRDIFF_MIN
ptrdiff_t

PTRDIFF_WIDTH
putc
putchar
puts
putwc
putwchar
qsort
qsort_s
quick_exit
raise
rand
RAND_MAX
realloc
register
remainder
remainderf
remainderl
remove
remquo
remquof
remquol
rename
restrict
return
rewind
rint
rintf
rintl
round
roundeven
roundevenf
roundevenl
roundf
roundl
RSIZE_MAX
rsize_t
scalbln
scalblnf
scalblnl
scalbn
scalbnf
scalbnl
scanf
scanf_s
SCHAR_MAX
SCHAR_MIN
SCHAR_WIDTH
SEEK_CUR
SEEK_END
SEEK_SET
setbuf
set_constraint_handler_s
setjmp
setlocale
setpayload
setpayloadf
setpayloadl
setpayloadsig
setpayloadsigf
setpayloadsigl
setvbuf
short
SHRT_MAX
SHRT_MIN
SHRT_WIDTH
sig_atomic_t
signal
signbit
signed
sin
sinf
sinh
sinhf
sinhl
sinl
SIZE_MAX
sizeof
size_t
SIZE_WIDTH
SNAN
SNANF
SNANL
snprintf
snprintf_s
snwprintf_s
sprintf
sprintf_s
sqrt
sqrtf
sqrtl
srand
sscanf
sscanf_s
static
static_assert
STDC
stderr
stdin
stdout
switch
swprintf
swprintf_s
swscanf
swscanf_s
system
tan
tanf
tanh
tanhf
tanhl
tanl
tgamma
tgammaf
tgammal
thousands_sep
thread_local
time
timespec
timespec_get
time_t
tm
tm_hour
tm_isdst
tm_mday
tm_min
tm_mon
tmpfile
tmpfile_s
TMP_MAX
TMP_MAX_S
tmpnam
tmpnam_s
tm_sec
tm_wday
tm_yday
tm_year
true
trunc
truncf
truncl
TSS_DTOR_ITERATIONS
tv_nsec
tv_sec
typedef
UCHAR_MAX
UCHAR_WIDTH
~~ufromfp~~
~~ufromfpf~~
~~ufromfpl~~
~~ufromfpx~~
~~ufromfpxf~~
~~ufromfpxl~~
ULLONG_MAX
ULLONG_WIDTH
ULONG_MAX
ULONG_WIDTH
undef
ungetc
ungetwc
union
unsigned
USHRT_MAX
USHRT_WIDTH
va_arg
va_copy
va_end
va_list
va_start

vfprintf
vfprintf_s
vfscanf
vfscanf_s
vfwprintf
vfwprintf_s
vfwscanf
vfwscanf_s
void
volatile
vprintf
vprintf_s
vscanf
vscanf_s
vsnprintf
vsnprintf_s
vsnwprintf_s
vsprintf
vsprintf_s
vsscanf
vsscanf_s
vswprintf
vswprintf_s
vswscanf
vswscanf_s
vwprintf
vwprintf_s
vwscanf
vwscanf_s
WCHAR_MAX
WCHAR_MIN
wchar_t
WCHAR_WIDTH
wcrtomb
wcrtomb_s
wctob
wctomb
wctomb_s
wctrans
wctrans_t
wctype
wctype_t
WEOF
while
WINT_MAX
WINT_MIN
wint_t
WINT_WIDTH
wmemchr
wmemcmp
wmemcpy
wmemcpy_s
wmemmove
wmemmove_s
wmemset
wprintf
wprintf_s
wscanf
wscanf_s
xor
xor_eq