ISO/IEC JTC 1/SC 22/WG14 **N2412**

**v 2**

August 11, 2019

# Two's complement sign representation for C2x

## Modification request for C2x

JF Bastien and Jens Gustedt
Apple Inc., USA, and INRIA and ICube, Université de Strasbourg, France

We implement the agreed change to abandon ones' complement and sign-and-magnitude representation from C.

This is a follow-up to document **N2218**[1] which found positive WG14 support to make two's complement the only sign representation for the next C standard, and a follow-up to document **N2330**[2] which only generated partial consensus in the London 2019 meeting of WG14.

## 1. INTRODUCTION

Removing other sign representations than two's complement from C allows to simplify the specification of integer types substantially. As has been voted in the London 2019 meeting we only implement the essential changes to the sign representation in this paper, namely

— Remove the specifications of the other sign representations.
— Impose the value of the minimal value of a signed type to be $-2^{N-1}$ where $N$ is the width of the type.
— Derive the values of the `_MIN` and `_MAX` macros for all integer types from the corresponding `_WIDTH` macro.
— Clean-up the remaining parts of the standard from all obsolete mentions of two's complement and negative integer zeros.

WG21 has recently adapted the changes promoted in their document **p1236**[3]. Generally, C++ goes much beyond what is presented here:

— Bit-fields can have excess bits.
— Overflowing operations and out-of-range conversion are generally mapped to modulo operations and cannot trap or raise signals.
— Enumeration types and their underlying compatible integer types have precise definitions.

WG14 has not yet found consensus for these points, so we leave them as they are in the current specification.

## 2. TWO'S COMPLEMENT

Restricting the possible sign representations to two's complement is relatively straight forward and does not need much of deep thinking.
There are some other direct fallouts from doing this, such as other mentions of two's complement in the document that now become obsolete. This concerns in particular the definition of the exact width integer types, and of the (bogus) specifications of arithmetic on atomic types.

## 3. TIGHTENING OF INTEGER REPRESENTATIONS

### 3.1. Minimum values of signed integer types

Even for two's complement representation C17 allowed that the value with sign bit 1 and all other bits 0 might be a trap representation. We change this and are thereby in line with

[1] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2218.htm
[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2330.pdf
[3] http://wg21.link/p1236

the changes in C++. We force that for integer types with a width of $N$ the minimum value is forced to $-2^{N-1}$ (and the maximum value remains at $2^{N-1}-1$).

### 3.2. Adjust widths of signed and unsigned integer types

In C17, the widths of corresponding signed and unsigned may differ by one, in particular an unsigned may be realized by just masking out the sign bit of the signed type. This possibility does not seem to be used in the field, complicates arguing about integers and adds potential case analysis to programs.

## 4. WIDTH, MINIMUM AND MAXIMUM MACROS FOR INTEGER TYPES

With these aggreed changes the relationship between the unsigned maximum and signed minimum and maximum values now becomes much simpler and can easily be expressed through the widths of the types, namely if the width is $N$ these values are now fixed to $2^N-1$, $-2^{N-1}$ and $2^{N-1}-1$, respectively. Since the integration of the floating point TS's already brings in macros that specify the width of the standard integer types, we simplify the presentation such that it is centered around the width.

This has the advantage that all requirements for the minimum width of integer types can now be presented as requirements of **_WIDTH** macros, and the specification of the **_MIN** and **_MAX** can be generic.

## 5. PROPOSED TEXT

As usual, we provide a diff-marked set of changed pages in an appendix. Unfortunately, for the central parts of the proposed changes the diff-marked text is not very readable so we provide the whole text for 5.2.4.2.1, 6.2.6.2, 7.20p5, 7.20.2, and 7.20.3, and only the changes that are textually small are reported in the diffmark appendix.

#### 5.2.4.2.1 Characteristics of integer types <limits.h>

1 The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. Their implementation-defined values shall be equal or greater to those shown.

— width for an object of type **_Bool**

```
BOOL_WIDTH                    1
```

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

```
CHAR_BIT                      8
```

The macros **CHAR_WIDTH**, **SCHAR_WIDTH**, and **UCHAR_WIDTH** that represent the width of the types **char**, **signed char** and **unsigned char** shall expand to the same value as **CHAR_BIT**.

— width for an object of type **unsigned short int**

```
USHRT_WIDTH                  16
```

The macro **SHRT_WIDTH** represents the width of the type **short int** and shall expand to the same value as **USHRT_WIDTH**.

— width for an object of type **unsigned int**

```
UINT_WIDTH                   16
```

The macro **INT_WIDTH** represents the width of the type **int** and shall expand to the same value as **UINT_WIDTH**.

— width for an object of type **unsigned long int**

```
ULONG_WIDTH                    32
```

The macro **LONG_WIDTH** represents the width of the type **long int** and shall expand to the same value as **ULONG_WIDTH**.

— width for an object of type **unsigned long long int**

```
ULLONG_WIDTH                   64
```

The macro **LLONG_WIDTH** represents the width of the type **long long int** and shall expand to the same value as **ULLONG_WIDTH**.

— maximum number of bytes in a multibyte character, for any supported locale

```
MB_LEN_MAX                     1
```

2 For all unsigned integer types for which <limits.h> or <stdint.h> define a macro with suffix **_WIDTH** holding its width $N$, there is a macro with suffix **_MAX** holding the maximal value $2^N - 1$ that is representable by the type, that is suitable for use in **#if** preprocessing directives and that has the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

3 For all signed integer types for which <limits.h> or <stdint.h> define a macro with suffix **_WIDTH** holding its width $N$, there are macros with suffix **_MIN** and **_MAX** holding the minimal and maximal values $-2^{N-1}$ and $2^{N-1} - 1$ that are representable by the type, that are suitable for use in **#if** preprocessing directives and that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

4 If an object of type **char** can hold negative values, the value of **CHAR_MIN** shall be the same as that of **SCHAR_MIN** and the value of **CHAR_MAX** shall be the same as that of **SCHAR_MAX**. Otherwise, the value of **CHAR_MIN** shall be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.[4]

...

#### 6.2.6.2 Integer types

1 For unsigned integer types the bits of the object representation shall be divided into two groups: value bits and padding bits. If there are $N$ value bits, each bit shall represent a different power of 2 between 1 and $2^{N-1}$, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified.The number of value bits $N$ is called the *width* of the unsigned integer type. There need not be any padding bits; **unsigned char** shall not have any padding bits.

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. If the corresponding unsigned type has width $N$, the signed type uses the same number of $N$ bits, its *width*, as value bits and sign bit. $N-1$ are value bits and the remaining bit is the sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type. If the sign bit

[4] See 6.2.5.

is zero, it shall not affect the resulting value. If the sign bit is one, it has value $-(2^{N-1})$. There need not be any padding bits; **signed char** shall not have any padding bits.

3 The values of any padding bits are unspecified. A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

4 The *precision* of an integer type is the number of value bits.

NOTE 1. *Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.*

NOTE 2. *The sign representation defined in this document is called* two's complement*. Previous revisions of this document additionally allowed other sign representations.*

NOTE 3. *For unsigned integer types the width and precision are the same, while for signed integer types the width is one greater than the precision.*

...

### 7.20 Integer types <stdint.h>

...

5 For all integer types for which there is a macro with suffix **_WIDTH** holding the width, maximum macros with suffix **_MAX** and, for all signed types, minimum macros with suffix **_MIN** are defined as by 5.2.4.2.

...

#### 7.20.2 Widths of specified-width integer types

1 The following object-like macros specify the width of the types declared in <stdint.h>. Each macro name corresponds to a similar type name in 7.20.1.

2 Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives. Its implementation-defined value shall be equal to or greater than the value given below, except where stated to be exactly the given value. An implementation shall define only the macros corresponding to those typedef names it actually provides.[5]

##### 7.20.2.1 Width of exact-width integer types

1

| | |
|---|---|
| **INTN_WIDTH** | exactly $N$ |
| **UINTN_WIDTH** | exactly $N$ |

##### 7.20.2.2 Width of minimum-width integer types

[5] The exact-width and pointer-holding integer types are optional.

| | |
|---|---|
| `INT_LEASTN_WIDTH` | exactly `UINT_LEASTN_WIDTH` |
| `UINT_LEASTN_WIDTH` | $N$ |

#### 7.20.2.3 Width of fastest minimum-width integer types

1

| | |
|---|---|
| `INT_FASTN_WIDTH` | exactly `UINT_FASTN_WIDTH` |
| `UINT_FASTN_WIDTH` | $N$ |

#### 7.20.2.4 Width of integer types capable of holding object pointers

1

| | |
|---|---|
| `INTPTR_WIDTH` | exactly `UINTPTR_WIDTH` |
| `UINTPTR_WIDTH` | 16 |

#### 7.20.2.5 Width of greatest-width integer types

1

| | |
|---|---|
| `INTMAX_WIDTH` | exactly `UINTMAX_WIDTH` |
| `UINTMAX_WIDTH` | 64 |

### 7.20.3 Characteristics of other integer types

1 The following object-like macros specify the width of integer types corresponding to types defined in other standard headers. If it is unspecified if a type is signed or unsigned and the implementation has it as an unsigned type, a minimum macro with extension `_MIN`, and value `0` of the corresponding type is defined.

2 Each instance of these macros shall be replaced by a constant expression suitable for use in `#if` preprocessing directives. Its implementation-defined value shall be equal to or greater than the corresponding value given below. An implementation shall define only the macros corresponding to those typedef names it actually provides.[6]

#### 7.20.3.1 Width of `ptrdiff_t`

1

| | |
|---|---|
| `PTRDIFF_WIDTH` | 17 |

#### 7.20.3.2 Width of `sig_atomic_t`

1

| | |
|---|---|
| `SIG_ATOMIC_WIDTH` | 8 |

#### 7.20.3.3 Width of `size_t`

1

| | |
|---|---|
| `SIZE_WIDTH` | 16 |

[6] A freestanding implementation need not provide all of these types.

#### 7.20.3.4 Width of `wchar_t`

```
1 WCHAR_WIDTH 8
```

#### 7.20.3.5 Width of `wint_t`

```
1 WINT_WIDTH 16
```

## Appendix: pages with diffmarks of the proposed changes

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

# Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

2 The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

3 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

4 Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

5 For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see the following URL: www.iso.org/iso/foreword.html.

6 This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

7 This fifth edition cancels and replaces the fourth edition, ISO/IEC 9899:2018. Major changes from the previous edition include:

— remove obsolete sign representations and integer width constraints

— added a one-argument version of **`_Static_assert`**

— harmonization with ISO/IEC 9945 (POSIX):

- extended month name formats for **`strftime`**
- integration of functions: **`memccpy`**, **`strdup`**, **`strndup`**

— harmonization with floating point standard IEC 60559:

- integration of binary floating-point technical specification TS 18661-1
- integration of decimal floating-point technical specification TS 18661-2
- integration of decimal floating-point technical specification TS 18661-4a

— the macro **`DECIMAL_DIG`** is declared obsolescent

— added version test macros to certain library headers

— added the attributes feature

— added **`nodiscard`**, **`maybe_unused`** and **`deprecated`** attributes

8 A complete change history can be found in Annex M.

— 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 Characteristics of integer types `<limits.h>`

1 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.~~ .

— width for an object of type **`_Bool`**

```
BOOL_WIDTH                  1
```

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

— 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**~~

The macros **CHAR_WIDTH**, **SCHAR_WIDTH**, and **UCHAR_WIDTH** that represent the width of the types **char**, **signed char** and **unsigned char** shall expand to the same value as **CHAR_BIT**.

— ~~maximum value~~ width for an object of type ~~**signed char**~~ **unsigned short int**

| | |
|---|---|
| ~~SCHAR_MAX~~ | ~~+127 // $2^7 - 1$~~ |
| USHRT_WIDTH | 16 |

~~width of type **signed char**~~

The macro **SHRT_WIDTH** represents the width of the type **short int** and shall expand to the same value as **USHRT_WIDTH**.

— ~~maximum value~~ width for an object of type ~~**unsigned char**~~ **unsigned int**

| | |
|---|---|
| ~~UCHAR_MAX~~ | ~~255 // $2^8 - 1$~~ |
| UINT_WIDTH | 16 |

~~width of type **unsigned char**~~

The macro **INT_WIDTH** represents the width of the type **int** and shall expand to the same value as **UINT_WIDTH**.

— ~~minimum value~~ width for an object of type ~~**char**~~ **unsigned long int**

| | |
|---|---|
| ~~CHAR_MIN~~ | ~~*see below*~~ |
| ULONG_WIDTH | 32 |

The macro **LONG_WIDTH** represents the width of the type **long int** and shall expand to the same value as **ULONG_WIDTH**.

— ~~maximum value~~ width for an object of type ~~**char**~~ **unsigned long long int**

| | |
|---|---|
| ~~CHAR_MAX~~ | ~~*see below*~~ |
| ULLONG_WIDTH | 64 |

~~width of type **char**~~

The macro **LLONG_WIDTH** represents the width of the type **long long int** and shall expand to the same value as **ULLONG_WIDTH**.

— maximum number of bytes in a multibyte character, for any supported locale

| | |
|---|---|
| **MB_LEN_MAX** | 1 |

~~minimum value for an object of type **short int**~~

~~maximum value for an object of type **short int**~~

~~width of type **short int**~~

2

~~maximum value for~~ For all unsigned integer types for which <limits.h> or <stdint.h> define a macro with suffix **_WIDTH** holding its width $N$, there is a macro with suffix **_MAX** holding the maximal value $2^N - 1$ that is representable by the type, that is suitable for use in **#if** preprocessing directives and that has the same type as would an expression that is an object of ~~type **unsigned short int**~~

~~width of type **unsigned short int**~~

~~minimum value for an object of type **int**~~

the corresponding type converted according to the integer promotions. ~~maximum value for an object of type **int**~~

~~width of type **int**~~

~~maximum value for an object of type **unsigned int**~~

~~width of type**unsigned int**~~

~~minimum value for an object of type **long int**~~

3

~~maximum value for~~ For all signed integer types for which <limits.h> or <stdint.h> define a macro with suffix **_WIDTH** holding its width $N$, there are macros with suffix **_MIN** and **_MAX** holding the minimal and maximal values $-2^{N-1}$ and $2^{N-1} - 1$ that are representable by the type, that are suitable for use in **#if** preprocessing directives and that have the same type as would an expression that is an object of ~~type **long int**~~

the corresponding type converted according to the integer promotions.

~~width of type **long int**~~

~~maximum value for an object of type **unsigned long int**~~

~~width of type **unsigned long int**~~

~~minimum value for an object of type **long long int**~~

~~maximum value for an object of type **long long int**~~

~~width of type **long long int**~~

~~maximum value for an object of type **unsigned long long int**~~

~~width of type **unsigned long long int**~~

4 If an object of type **char** can hold negative values, the value of **CHAR_MIN** shall be the same as that of **SCHAR_MIN** and the value of **CHAR_MAX** shall be the same as that of **SCHAR_MAX**. Otherwise, the value of **CHAR_MIN** shall be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.[20] ~~The value **UCHAR_MAX** shall equal $2^{\texttt{CHAR\_BIT}} - 1$.~~

**Forward references:** representations of types (6.2.6), conditional inclusion (6.10.1), integer types <stdint.h> (7.20).

#### 5.2.4.2.2 Characteristics of floating types <float.h>

1 The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[21] An implementation that defines **__STDC_IEC_60559_BFP__** or **__STDC_IEC_559__** shall implement floating point types and arithmetic conforming to IEC 60559 as specified in Annex F. An implementation that defines **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__** shall implement complex types and arithmetic conforming to

[20] See 6.2.5.

[21] The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical.

32 **EXAMPLE 2** The type designated as "`struct tag (*[5])(float)`" has type "array of pointer to function returning `struct tag`". The array has length five and the function has a single parameter of type `float`. Its type category is array.

**Forward references:** compatible type and composite type (6.2.7), declarations (6.7).

## 6.2.6 Representations of types

### 6.2.6.1 General

1 The representations of all types are unspecified except as stated in this subclause.

2 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.

3 Values stored in unsigned bit-fields and objects of type `unsigned char` shall be represented using a pure binary notation.[53)]

4 Values stored in non-bit-field objects of any other object type consist of $n \times$ `CHAR_BIT` bits, where $n$ is the size of an object of that type, in bytes. The value may be copied into an object of type `unsigned char [`$n$`]` (e.g., by `memcpy`); the resulting set of bytes is called the *object representation* of the value. Values stored in bit-fields consist of $m$ bits, where $m$ is the size specified for the bit-field. The object representation is the set of $m$ bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

5 Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[54)] Such a representation is called a trap representation.

6 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[55)] The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

7 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

8 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[56)] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

9 Loads and stores of objects with atomic types are done with `memory_order_seq_cst` semantics.

**Forward references:** declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3).

### 6.2.6.2 Integer types

1 For unsigned integer types ~~other than `unsigned char`,~~ the bits of the object representation shall be divided into two groups: value bits and padding bits. ~~(there need not be any of the latter).~~ If there are $N$ value bits, each bit shall represent a different power of 2 between 1 and $2^{N-1}$, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits

[53)] A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.) A byte contains `CHAR_BIT` bits, and the values of type `unsigned char` range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$.

[54)] Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[55)] Thus, for example, structure assignment need not copy any padding bits.

[56)] It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts. In particular, if == is defined for type `T`, then `x == y` does not imply that `memcmp(&x, &y, sizeof (T))== 0`. Furthermore, `x == y` does not necessarily imply that `x` and `y` have the same value; other operations on values of type `T` might distinguish between them.

are unspecified.The number of value bits $N$ is called the *width* of the unsigned integer type. There need not be any padding bits; **unsigned char** shall not have any padding bits.

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. ~~There need not be any padding bits; **signed char** shall not have any padding bits . There shall be exactly one~~ If the corresponding unsigned type has width $N$, the signed type uses the same number of $N$ bits, its *width* , as value bits and sign bit. $N - 1$ are value bits and the remaining bit is the sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type~~(if there are $M$ value bits in the signed type and $N$ in the unsigned type, then $M \leq N$).~~ . If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, ~~the value shall be modified in one of the following ways:~~

~~the corresponding value with sign bit 0 is negated (); the sign bit has the value $-(2^M)$ (); the sign bit has the value $-(2^M - 1)$ ().~~

~~Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value . In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a .~~

~~If the implementation supports negative zeros, they shall be generated only by: the &, |, ^, ~, <<, and >> operators with operands that produce such a value; the +, - , *, /, and %~~ it has value $-(2^{N-1})$. ~~operators where one operand is a negative zero and the result is zero; compound assignment operators based on the above cases. It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.~~

~~If the implementation does not support negative zeros, the behavior of the &, |, ^, ~, <<, and >> operators with operands that would produce such a value is undefined~~There need not be any padding bits; **signed char** shall not have any padding bits.

3 The values of any padding bits are unspecified. A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

4 The *precision* of an integer type is the number of ~~bits it uses to represent values , excluding any sign and padding bits~~value bits. ~~The of an integer type is the same but including any sign bit; thus for~~

5 **NOTE 1** Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

6 **NOTE 2** The sign representation defined in this document is called *two's complement* . Previous revisions of this document additionally allowed other sign representations.

7 **NOTE 3** For unsigned integer types the ~~two values~~ width and precision are the same, while for signed integer types the width is one greater than the precision.

### 6.2.7 Compatible type and composite type

1 Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators.[57] Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are completed anywhere within their respective translation units, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types; if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier; and if one member of the pair is declared with a name, the other is declared with the same name. For

[57] Two types need not be identical to be compatible.

construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

6 The hexadecimal digits that follow the backslash and the letter `x` in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

7 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

8 In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash \ followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.[81)]

**Constraints**

9 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the corresponding type:

| Prefix | Corresponding Type |
|---|---|
| none | **`unsigned char`** |
| **`L`** | the unsigned type corresponding to **`wchar_t`** |
| **`u`** | **`char16_t`** |
| **`U`** | **`char32_t`** |

**Semantics**

10 An integer character constant has type **`int`**. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., `'ab'`), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **`char`** whose value is that of the single character or escape sequence is converted to type **`int`**.

11 A wide character constant prefixed by the letter **`L`** has type **`wchar_t`**, an integer type defined in the `<stddef.h>` header; a wide character constant prefixed by the letter **`u`** or **`U`** has type **`char16_t`** or **`char32_t`**, respectively, unsigned integer types defined in the `<uchar.h>` header. The value of a wide character constant containing a single multibyte character that maps to a single member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **`mbtowc`**, **`mbrtoc16`**, or **`mbrtoc32`** function as appropriate for its type, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

12 **EXAMPLE 1** The construction `'\0'` is commonly used to represent the null character.

13 **EXAMPLE 2** Consider implementations that use ~~two's complement representation for integers and~~ eight bits for objects that have type **`char`**. In an implementation in which type **`char`** has the same range of values as **`signed char`**, the integer character constant `'\xFF'` has the value −1; if type **`char`** has the same range of values as **`unsigned char`**, the character constant `'\xFF'` has the value +255.

14 **EXAMPLE 3** Even if eight bits are used for objects that have type **`char`**, the construction `'\x123'` specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are `'\x12'` and `'3'`, the construction `'\0223'` can be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)

15 **EXAMPLE 4** Even if 12 or more bits are used for objects that have type **`wchar_t`**, the construction `L'\1234'` specifies the implementation-defined value that results from the combination of the values `0123` and `'4'`.

[81)] The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See "future language directions" (6.11.4).

7 **EXAMPLE** A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

```
exp = atomic_load(&cur);
do {
      des = function(exp);
} while (!atomic_compare_exchange_weak(&cur, &exp, des));
```

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

**Returns**

8 The result of the comparison.

### 7.17.7.5 The `atomic_fetch` and modify generic functions

1 The following operations perform arithmetic and bitwise computations. All of these operations are applicable to an object of any atomic integer type. None of these operations is applicable to **`atomic_bool`**. The key, operator, and computation correspondence is:

| key | op | computation |
|---|---|---|
| add | + | addition |
| sub | - | subtraction |
| or | \| | bitwise inclusive or |
| xor | ^ | bitwise exclusive or |
| and | & | bitwise and |

**Synopsis**

2
```
#include <stdatomic.h>
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
```

**Description**

3 Atomically replaces the value pointed to by `object` with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (5.1.2.4). For signed integer types, arithmetic is defined to use ~~two's complement representation with~~ silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

**Returns**

4 Atomically, the value pointed to by `object` immediately before the effects.

5 **NOTE** The operation of the `atomic_fetch` and modify generic functions are nearly equivalent to the operation of the corresponding *op*= compound assignment operators. The only differences are that the compound assignment operators are not guaranteed to operate atomically, and the value yielded by a compound assignment operator is the updated value of the object, whereas the value returned by the `atomic_fetch` and modify generic functions is the previous value of the atomic object.

## 7.17.8 Atomic flag type and operations

1 The **`atomic_flag`** type provides the classic test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type **`atomic_flag`** shall be lock free.

3 **NOTE** Hence, as per 7.17.5, the operations should also be address-free. No other type requires lock-free operations, so the **`atomic_flag`** type is the minimum hardware-implemented type needed to conform to this document. The remaining types can be emulated with **`atomic_flag`**, though with less than ideal properties.

4 The macro **`ATOMIC_FLAG_INIT`** may be used to initialize an **`atomic_flag`** to the clear state. An **`atomic_flag`** that is not explicitly initialized with **`ATOMIC_FLAG_INIT`** is initially in an indeterminate state.

5 **EXAMPLE**

## 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.[279] 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,[280] <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 feature test macro **__STDC_VERSION_STDINT_H__** 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* ~~,~~ and 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.

2 The typedef name **uint***N***_t** designates an unsigned integer type with width *N* and no padding bits. Thus, uint24_t denotes such an unsigned integer type with a width of exactly 24 bits.

3 These types are optional. However, if an implementation provides integer types with widths of 8, 16, 32, or 64 bits, and no padding bits, ~~and (for the signed types) that have a two's complement representation,~~ it shall define the corresponding typedef names.

#### 7.20.1.2 Minimum-width integer types

1 The typedef name **int_least***N***_t** designates a signed integer type with a width of at least *N*, such that no signed integer type with lesser size has at least the specified width. Thus, **int_least32_t** denotes a signed integer type with a width of at least 32 bits.

2 The typedef name **uint_least***N***_t** designates an unsigned integer type with a width of at least *N*, such that no unsigned integer type with lesser size has at least the specified width. Thus, **uint_least16_t** denotes an unsigned integer type with a width of at least 16 bits.

3 The following types are required:

[279] See "future library directions" (7.31.12).
[280] Some of these types might denote implementation-defined extended integer types.

```
int_least8_t                    uint_least8_t
int_least16_t                   uint_least16_t
int_least32_t                   uint_least32_t
int_least64_t                   uint_least64_t
```

All other types of this form are optional.

#### 7.20.1.3 Fastest minimum-width integer types

1 Each of the following types designates an integer type that is usually fastest[281)] to operate with among all integer types that have at least the specified width.

2 The typedef name **int_fast***N*_t designates the fastest signed integer type with a width of at least *N*. The typedef name **uint_fast***N*_t designates the fastest unsigned integer type with a width of at least *N*.

3 The following types are required:

```
int_fast8_t                     uint_fast8_t
int_fast16_t                    uint_fast16_t
int_fast32_t                    uint_fast32_t
int_fast64_t                    uint_fast64_t
```

All other types of this form are optional.

#### 7.20.1.4 Integer types capable of holding object pointers

1 The following type designates a signed integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
intptr_t
```

The following type designates an unsigned integer type with the property that any valid pointer to **void** can be converted to this type, then converted back to pointer to **void**, and the result will compare equal to the original pointer:

```
uintptr_t
```

These types are optional.

#### 7.20.1.5 Greatest-width integer types

1 The following type designates a signed integer type capable of representing any value of any signed integer type:

```
intmax_t
```

The following type designates an unsigned integer type capable of representing any value of any unsigned integer type:

```
uintmax_t
```

These types are required.

### 7.20.2 Widths of specified-width integer types

1 The following object-like macros specify the ~~minimum and maximum limits~~ width of the types declared in <stdint.h>. Each macro name corresponds to a similar type name in 7.20.1.

[281)] The designated type is not guaranteed to be fastest for all purposes; if the implementation has no clear grounds for choosing one type over another, it will simply pick some integer type satisfying the signedness and width requirements.

2 Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives~~, and, except for the width-of-type macros, this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.~~. Its implementation-defined value shall be equal to or greater ~~in magnitude (absolute value) than the corresponding~~ than the value given below, ~~with the same sign,~~ except where stated to be exactly the given value.

~~minimum values of exact-width signed integer types maximum values of exact-width signed integer types maximum values of exact-width unsigned integer types~~

~~width of exact-width signed integer types~~

~~width of exact-width unsigned integer types~~ An implementation shall define only the macros corresponding to those typedef names it actually provides.[282)]

### 7.20.2.1 Width of exact-width integer types

~~minimum values of minimum-width signed integer types maximum values of minimum-width signed integer types maximum values of minimum-width unsigned integer types~~

1

| | |
|---|---|
| **INT*N*_WIDTH** | exactly $N$ |
| **UINT*N*_WIDTH** | exactly $N$ |

~~width of minimum-width signed integer types~~

### 7.20.2.2 Width of minimum-width integer types

~~width of minimum-width unsigned integer types~~

1

| | |
|---|---|
| **INT_LEAST*N*_WIDTH** | exactly **UINT_LEAST*N*_WIDTH** |
| **UINT_LEAST*N*_WIDTH** | $N$ |

### 7.20.2.3 Width of fastest minimum-width integer types

~~minimum values of fastest minimum-width signed integer types maximum values of fastest minimum-width signed integer types maximum values of fastest minimum-width unsigned integer types~~

1

| | |
|---|---|
| **INT_FAST*N*_WIDTH** | exactly **UINT_FAST*N*_WIDTH** |
| **UINT_FAST*N*_WIDTH** | $N$ |

~~width of fastest minimum-width signed integer types~~

~~width of fastest minimum-width unsigned integer types~~

### 7.20.2.4 Width of integer types capable of holding object pointers

~~minimum value of pointer-holding signed integer type~~

1

| | |
|---|---|
| ~~INTPTR_MIN~~ | ~~$-(2^{15}-1)$~~ |
| INTPTR_WIDTH | exactly UINTPTR_WIDTH |
| UINTPTR_WIDTH | 16 |

~~maximum value of pointer-holding signed integer type~~ ~~maximum value of pointer-holding unsigned integer type~~

~~width of pointer-holding signed integer type~~ ~~width of pointer-holding unsigned integer type~~

### 7.20.2.5 Width of greatest-width integer types

~~minimum value of greatest-width signed integer type~~

1 [282)] The exact-width and pointer-holding integer types are optional.

| | |
|---|---|
| **INTMAX_WIDTH** | exactly **UINTMAX_WIDTH** |
| **UINTMAX_WIDTH** | 64 |

~~maximum value of greatest-width signed integer type~~ ~~maximum value of greatest-width unsigned integer type~~

~~width of greatest-width signed integer type~~ ~~width of greatest-width unsigned integer type~~

### 7.20.3 Width of other integer types

1 The following object-like macros specify the ~~minimum and maximum limits~~ width of integer types corresponding to types defined in other standard headers.

2 Each instance of these macros shall be replaced by a constant expression suitable for use in **#if** preprocessing directives~~, and this expression shall have the same type, except for the width-of-type macros, as would an expression that is an object of the corresponding type converted according to the integer promotions.~~ . Its implementation-defined value shall be equal to or greater ~~in magnitude (absolute value)~~ than the corresponding value given below~~, with the same sign~~. An implementation shall define only the macros corresponding to those typedef names it actually provides.[283] ~~limits of~~

#### 7.20.3.1 Width of **ptrdiff_t**

1

| | |
|---|---|
| ~~PTRDIFF_MIN~~ | ~~-65535~~ |
| ~~PTRDIFF_MAX~~ | ~~+65535~~ |
| ~~PTRDIFF_WIDTH~~ | ~~16~~ |
| PTRDIFF_WIDTH | 17 |

~~limits of~~

#### 7.20.3.2 Width of **sig_atomic_t**

1

| | |
|---|---|
| ~~SIG_ATOMIC_MIN~~ | ~~*see below*~~ |
| ~~SIG_ATOMIC_MAX~~ | ~~*see below*~~ |
| **SIG_ATOMIC_WIDTH** | 8 |

~~limit of limits of limits of~~

~~If **sig_atomic_t** (see 7.14) is defined as a signed integer type, the value of **SIG_ATOMIC_MIN** shall be no greater than −127 and the value of **SIG_ATOMIC_MAX** shall be no less than 127; otherwise, **sig_atomic_t** is defined as an unsigned integer type, and the value of **SIG_ATOMIC_MIN** shall be 0 and the value of **SIG_ATOMIC_MAX** shall be no less than 255.~~

#### 7.20.3.3 Width of **size_t**

~~If **wchar_t** (see 7.19) is defined as a signed integer type, the value of **WCHAR_MIN** shall be no greater than −127 and the value of **WCHAR_MAX** shall be no less than 127; otherwise, **wchar_t** is defined as an unsigned integer type, and the value of **WCHAR_MIN** shall be 0 and the value of **WCHAR_MAX** shall be no less than 255.~~

1

| | |
|---|---|
| **SIZE_WIDTH** | 16 |

#### 7.20.3.4 Width of **wchar_t**

1

| | |
|---|---|
| **WCHAR_WIDTH** | 8 |

[283] A freestanding implementation need not provide all of these types.

#### 7.20.3.5 Width of `wint_t`

```
WINT_WIDTH                      16
```

~~If **wint_t** (see 7.29) is defined as a signed integer type, the value of **WINT_MIN** shall be no greater than −32767 and the value of **WINT_MAX** shall be no less than 32767; otherwise, **wint_t** is defined as an unsigned integer type, and the value of **WINT_MIN** shall be 0 and the value of **WINT_MAX** shall be no less than 65535.~~

### 7.20.4 Macros for integer constants

1 The following function-like macros expand to integer constants suitable for initializing objects that have integer types corresponding to types defined in <stdint.h>. Each macro name corresponds to a similar type name in 7.20.1.2 or 7.20.1.5.

2 The argument in any instance of these macros shall be an unsuffixed integer constant (as defined in 6.4.4.1) with a value that does not exceed the limits for the corresponding type.

3 Each invocation of one of these macros shall expand to an integer constant expression suitable for use in **#if** preprocessing directives. The type of the expression shall have the same type as would an expression of the corresponding type converted according to the integer promotions. The value of the expression shall be that of the argument.

#### 7.20.4.1 Macros for minimum-width integer constants

1 The macro **INT***N***_C**(*value*) expands to an integer constant expression corresponding to the type **int_least***N***_t**. The macro **UINT***N***_C**(*value*) expands to an integer constant expression corresponding to the type **uint_least***N***_t**. For example, if **uint_least64_t** is a name for the type **unsigned long long int**, then **UINT64_C**(0x123) might expand to the integer constant 0x123ULL.

#### 7.20.4.2 Macros for greatest-width integer constants

1 The following macro expands to an integer constant expression having the value specified by its argument and the type **intmax_t**:

```
INTMAX_C(value)
```

The following macro expands to an integer constant expression having the value specified by its argument and the type **uintmax_t**:

```
UINTMAX_C(value)
```

### 7.20.5 Maximal and minimal values of integer types

1 For all integer types for which there is a macro with suffix **_WIDTH** holding the width, maximum macros with suffix **_MAX** and, for all signed types, minimum macros with suffix **_MIN** are defined as by 5.2.4.2. If it is unspecified if a type is signed or unsigned and the implementation has it as an unsigned type, a minimum macro with extension **_MIN**, and value 0 of the corresponding type is defined.

## 7.29 Extended multibyte and wide character utilities <wchar.h>

### 7.29.1 Introduction

1 The header <wchar.h> defines four macros, and declares four data types, one tag, and many functions.[346)]

2 The types declared are **wchar_t** and **size_t** (both described in 7.19);

```
mbstate_t
```

which is a complete object type other than an array type that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters;

```
wint_t
```

which is an integer type unchanged by default argument promotions that can hold any value corresponding to members of the extended character set, as well as at least one value that does not correspond to any member of the extended character set (see **WEOF** below);[347)] and

```
struct tm
```

which is declared as an incomplete structure type (the contents are described in 7.27.1).

3 The macros defined are **NULL** (described in 7.19); **WCHAR_MIN**~~and~~ **WCHAR_MAX**, **WCHAR_MAX**, and **WCHAR_WIDTH** (described in 7.20 ); and

```
WEOF
```

which expands to a constant expression of type **wint_t** whose value does not correspond to any member of the extended character set.[348)] It is accepted (and returned) by several functions in this subclause to indicate *end-of-file*, that is, no more input from a stream. It is also used as a wide character value that does not correspond to any member of the extended character set.

4 The functions declared are grouped as follows:

— Functions that perform input and output of wide characters, or multibyte characters, or both;

— Functions that provide wide string numeric conversion;

— Functions that perform general wide string manipulation;

— Functions for wide string date and time conversion; and

— Functions that provide extended capabilities for conversion between multibyte and wide character sequences.

5 Arguments to the functions in this subclause may point to arrays containing **wchar_t** values that do not correspond to members of the extended character set. Such values shall be processed according to the specified semantics, except that it is unspecified whether an encoding error occurs if such a value appears in the format string for a function in 7.29.2 or 7.29.5 and the specified semantics do not require that value to be processed by **wcrtomb**.

6 Unless explicitly stated otherwise, if the execution of a function described in this subclause causes copying to take place between objects that overlap, the behavior is undefined.

### 7.29.2 Formatted wide character input/output functions

1 The formatted wide character input/output functions shall behave as if there is a sequence point after the actions associated with each specifier.[349)]

[346)] See "future library directions" (7.31.18).
[347)] **wchar_t** and **wint_t** can be the same integer type.
[348)] The value of the macro **WEOF** can differ from that of **EOF** and need not be negative.
[349)] The **fwprintf** functions perform writes to memory for the %n specifier.

# Annex E
(informative)
# Implementation limits

1 The contents of the header `<limits.h>` are given below~~, in alphabetical order. The minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign~~. The values shall all be constant expressions suitable for use in **`#if`** preprocessing directives. The components are described further in ~~??~~5.2.4.2.1.

2 For the following macros, the minimum values shown shall be replaced by implementation-defined values.

```
#define BOOL_WIDTH                  1
#define CHAR_BIT                    8
#define USHRT_WIDTH                16
#define UINT_WIDTH                 16
#define ULONG_WIDTH                32
#define ULLONG_WIDTH               64
#define MB_LEN_MAX                  1
```

3 For the following macros, the minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign that are deduced from the macros above as indicated.[378)]

```
#define BOOL_MAX                    1  // 2^BOOL_WIDTH − 1
#define CHAR_MAX    UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN            0 or SCHAR_MIN
#define INT_MAX                 +32767
#define INT_MIN                 -32767
#define LONG_MAX            +2147483647
#define LONG_MIN            -2147483647
#define LLONG_MAX  +9223372036854775807
#define LLONG_MIN  -9223372036854775807
#define MB_LEN_MAX                    1
#define SCHAR_MAX                  +127
#define SCHAR_MIN                  -127
#define SHRT_MAX                 +32767
#define SHRT_MIN                 -32767
#define UCHAR_MAX                  255
#define USHRT_MAX                65535
#define UINT_MAX                 65535
#define ULONG_MAX           4294967295
#define ULLONG_MAX 18446744073709551615
#define CHAR_WIDTH                  8  // CHAR_BIT
#define INT_MAX                +32767  // 2^(INT_WIDTH−1) − 1
#define INT_MIN                -32768  // −2^(INT_WIDTH−1)
#define INT_WIDTH                 16  // UINT_WIDTH
#define LONG_MAX           +2147483647  // 2^(LONG_WIDTH−1) − 1
#define LONG_MIN           -2147483648  // −2^(LONG_WIDTH−1)
#define LONG_WIDTH                32  // ULONG_WIDTH
#define LLONG_MAX  +9223372036854775807  // 2^(LLONG_WIDTH−1) − 1
#define LLONG_MIN  -9223372036854775808  // −2^(LLONG_WIDTH−1)
#define LLONG_WIDTH               64  // ULLONG_WIDTH
#define SCHAR_MAX                +127  // 2^(SCHAR_WIDTH−1) − 1
#define SCHAR_MIN                -128  // −2^(SCHAR_WIDTH−1)
#define SCHAR_WIDTH                 8  // CHAR_BIT
#define SHRT_MAX                +32767  // 2^(SHRT_WIDTH−1) − 1
```

[378)] For the minimum value of a signed integer type there is no expression consisting of a minus sign and a decimal literal of that same type. The numbers in the table are only given as indications for the values and do not represent suitable expressions to be used for these macros.

```
#define SHRT_MIN                  -32768  // -2^(SHRT_WIDTH-1)
#define UCHAR_MAX                   255  // 2^UCHAR_WIDTH - 1
#define UCHAR_WIDTH                   8  // CHAR_BIT
#define USHRT_MAX                 65535  // 2^USHRT_WIDTH - 1
#define UINT_MAX                  65535  // 2^UINT_WIDTH - 1
#define ULONG_MAX              4294967295  // 2^ULONG_WIDTH - 1
#define ULLONG_MAX  18446744073709551615  // 2^ULLONG_WIDTH - 1
```

4 The contents of the header <float.h> are given below. All integer values, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. The components are described further in 5.2.4.2.2 and 5.2.4.2.3.

5 The values given in the following list shall be replaced by implementation-defined expressions:

```
#define FLT_EVAL_METHOD
#define FLT_ROUNDS
#ifdef __STDC_IEC_60559_DFP__
#define DEC_EVAL_METHOD
#endif
```

6 The values given in the following list shall be replaced by implementation-defined constant expressions that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

```
#define DBL_DECIMAL_DIG              10
#define DBL_DIG                      10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP              +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP              -37
#define DBL_MIN_EXP
#define DECIMAL_DIG                  10
#define FLT_DECIMAL_DIG               6
#define FLT_DIG                       6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP              +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP              -37
#define FLT_MIN_EXP
#define FLT_RADIX                     2
#define LDBL_DECIMAL_DIG             10
#define LDBL_DIG                     10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP             +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP             -37
#define LDBL_MIN_EXP
```

7 The values given in the following list shall be replaced by implementation-defined constant expressions with values that are greater than or equal to those shown:

```
#define DBL_MAX                   1E+37
#define DBL_NORM_MAX              1E+37
#define FLT_MAX                   1E+37
#define FLT_NORM_MAX              1E+37
#define LDBL_MAX                  1E+37
#define LDBL_NORM_MAX             1E+37
```

8 The values given in the following list shall be replaced by implementation-defined constant expressions with (positive) values that are less than or equal to those shown:

# Annex J
(informative)
# Portability issues

1 This annex collects some information about portability that appears in this document.

## J.1 Unspecified behavior

1 The following are unspecified:

— The manner and timing of static initialization (5.1.2).

— The termination status returned to the hosted environment if the return type of `main` is not compatible with int (5.1.2.2.3).

— The values of objects that are neither lock-free atomic objects nor of type `volatile sig_atomic_t` and the state of the floating-point environment, when the processing of the abstract machine is interrupted by receipt of a signal (5.1.2.3).

— The behavior of the display device if a printing character is written when the active position is at the final position of a line (5.2.2).

— The behavior of the display device if a backspace character is written when the active position is at the initial position of a line (5.2.2).

— The behavior of the display device if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (5.2.2).

— The behavior of the display device if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (5.2.2).

— How an extended source character that does not correspond to a universal character name counts toward the significant initial characters in an external identifier (5.2.4.1).

— Many aspects of the representations of types (6.2.6).

— The value of padding bytes when storing values in structures or unions (6.2.6.1).

— The values of bytes that correspond to union members other than the one last stored into (6.2.6.1).

— The representation used when storing a value in an object that has more than one object representation for that value (6.2.6.1).

— The values of any padding bits in integer representations (6.2.6.2).

— Whether ~~certain operators can generate negative zeros and whether a negative zero becomes a normal zero when stored in an object (6.2.6.2). Whether~~ two string literals result in distinct arrays (6.4.5).

— The order in which subexpressions are evaluated and the order in which side effects take place, except as specified for the function-call `()`, `&&`, `||`, `?:`, and comma operators (6.5).

— The order in which the function designator, arguments, and subexpressions within the arguments are evaluated in a function call (6.5.2.2).

— The order of side effects among compound literal initialization list expressions (6.5.2.5).

— The order in which the operands of an assignment operator are evaluated (6.5.16).

— The alignment of the addressable storage unit allocated to hold a bit-field (6.7.2.1).

— Whether a call to an inline function uses the inline definition or the external definition of the function (6.7.4).

## J.2 Undefined behavior

1 The behavior is undefined in the following circumstances:

— A "shall" or "shall not" requirement that appears outside of a constraint is violated (Clause 4).

— A nonempty source file does not end in a new-line character which is not immediately preceded by a backslash character or ends in a partial preprocessing token or comment (5.1.1.2).

— Token concatenation produces a character sequence matching the syntax of a universal character name (5.1.1.2).

— A program in a hosted environment does not define a function named **main** using one of the specified forms (5.1.2.2.1).

— The execution of a program contains a data race (5.1.2.4).

— A character not in the basic source character set is encountered in a source file, except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token (5.2.1).

— An identifier, comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state (5.2.1.2).

— The same identifier has both internal and external linkage in the same translation unit (6.2.2).

— An object is referred to outside of its lifetime (6.2.4).

— The value of a pointer to an object whose lifetime has ended is used (6.2.4).

— The value of an object with automatic storage duration is used while it is indeterminate (6.2.4, 6.7.9, 6.8).

— A trap representation is read by an lvalue expression that does not have character type (6.2.6.1).

— A trap representation is produced by a side effect that modifies any part of the object using an lvalue expression that does not have character type (6.2.6.1).

— ~~The operands to certain operators are such that they could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).~~ Two declarations of the same object or function specify types that are not compatible (6.2.7).

— A program requires the formation of a composite type from a variable length array type whose size is specified by an expression that is not evaluated (6.2.7).

— Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

— Demotion of one real floating type to another produces a value outside the range that can be represented (6.3.1.5).

— An lvalue does not designate an object when evaluated (6.3.2.1).

— A non-array lvalue with an incomplete type is used in a context that requires the value of the designated object (6.3.2.1).

— An lvalue designating an object of automatic storage duration that could have been declared with the **register** storage class is used in a context that requires the value of the designated object, but the object is uninitialized. (6.3.2.1).

— An lvalue having array type is converted to a pointer to the initial element of the array, and the array object has register storage class (6.3.2.1).

— An attempt is made to use the value of a void expression, or an implicit or explicit conversion (except to **void**) is applied to a void expression (6.3.2.2).

### J.3.3 Identifiers

1 — Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

— The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

### J.3.4 Characters

1 — The number of bits in a byte (3.6).

— The values of the members of the execution character set (5.2.1).

— The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

— The value of a **char** object into which has been stored any character other than a member of the basic execution character set (6.2.5).

— Which of **signed char** or **unsigned char** has the same range, representation, and behavior as "plain" **char** (6.2.5, 6.3.1.1).

— The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

— The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

— The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

— The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

— Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

— The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

— The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

— The encoding of any of **wchar_t**, **char16_t**, and **char32_t** where the corresponding standard encoding macro (**__STDC_ISO_10646__**, **__STDC_UTF_16__**, or **__STDC_UTF_32__**) is not defined (6.10.8.2).

### J.3.5 Integers

1 — Any extended integer types that exist in the implementation (6.2.5).

— ~~Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).~~ The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

— The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

— The results of some bitwise operations on signed integers (6.5).

### 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 38 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_]*
DEC128_[A-Z][a-zA-Z0-9_]*
DEC32_[A-Z][a-zA-Z0-9_]*
DEC64_[A-Z][a-zA-Z0-9_]*
DEC_[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_]*
LDBL_[A-Z][a-zA-Z0-9_]*
MATH_[A-Z][a-zA-Z0-9_]*
mem[a-z][a-zA-Z0-9_]*
mtx_[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 ~~629~~ 636 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
```

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
wcsto
wcstod
wcstod128
wcstod32
wcstod64
wcstof
wcstoimax
wcstok
wcstok_s
wcstol
wcstold
wcstoll
wcstombs
wcstombs_s
wcstoul
wcstoull
wcstoumax
wcsxfrm
~~_WIDTH~~

### J.6.2 Particular identifiers or keywords

1 The following ~~1188~~ 1190 identifiers or keywords are not covered by the above and have particular semantics provided by this document.

abort
abort_handler_s
abs
acos
acosd128
acosd32
acosd64
acosf
acosh
acoshd128
acoshd32
acoshd64
acoshf
acoshl
acosl
acospi
acospid128
acospid32
acospid64
acospif
acospil
alignas
aligned_alloc
alignof
and
and_eq
asctime
asctime_s
asin
asind128
asind32
asind64
asinf
asinh
asinhd128
asinhd32
asinhd64
asinhf
asinhl
asinl
asinpi
asinpid128
asinpid32
asinpid64
asinpif
asinpil
assert
atan
atan2
atan2d128
atan2d32
atan2d64
atan2f
atan2l
atan2pi
atan2pid128
atan2pid32
atan2pid64
atan2pif
atan2pil

# Annex M
(informative)
# Change History

## M.1 Fifth Edition

1 Major changes in this fifth edition (__STDC_VERSION__ *yyyymm*L) include:

— remove obsolete sign representations and integer width constraints

— added a one-argument version of **_Static_assert**

— harmonization with ISO/IEC 9945 (POSIX):

- extended month name formats for **strftime**
- integration of functions: **memccpy**, **strdup**, **strndup**

— harmonization with floating point standard IEC 60559:

- integration of binary floating-point technical specification TS 18661-1
- integration of decimal floating-point technical specification TS 18661-2
- integration of decimal floating-point technical specification TS 18661-4a

— the macro **DECIMAL_DIG** is declared obsolescent

— added version test macros to certain library headers

— added the attributes feature

— added **nodiscard**, **maybe_unused** and **deprecated** attributes

## M.2 Fourth Edition

1 There were no major changes in the fourth edition (__STDC_VERSION__ 201710L), only technical corrections and clarifications.

## M.3 Third Edition

1 Major changes in the third edition (__STDC_VERSION__ 201112L) included:

— conditional (optional) features (including some that were previously mandatory)

— support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (<stdatomic.h> and <threads.h>)

— additional floating-point characteristic macros (<float.h>)

— querying and specifying alignment of objects (<stdalign.h>, <stdlib.h>)

— Unicode characters and strings (<uchar.h>) (originally specified in ISO/IEC TR 19769:2004)

— type-generic expressions

— static assertions

— anonymous structures and unions

— no-return functions

— macros to create complex numbers (<complex.h>)

— support for opening files for exclusive access

— removed the **gets** function (<stdio.h>)