Moving to two's complement sign representation

Jean-François Bastien, Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2330**

January 15, 2019 **v 1**

Moving to two's complement sign representation

Modification request for C2x

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

We propose to implement the change to abandon ones complement and sign-and-magnitude representation from C. Main efforts are made to maintain interface compatibility with C++'s recently voted changes, that is to ensure that value and object representations are exchangeable.

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.

1. INTRODUCTION

Just banning ones complement and sign and magnitude representation is only part of what is necessary to ensure future interface interchangeability between C and C++. In fact, C and C++ had several other possibilities for integer representations and bit-fields that made arguing about integers tedious, but which are as obsolete and unused as are the exotic sign representations.

WG21 has recently adapted the changes promoted in their document **p1236**[2] that tighten these things up. Most of them concern interfaces, that is how integer data is represented and how functions might be called across the languages. We think that it would be wise for WG14 to follow the exact same changes.

Some other changes concern the operational side, such as conversions and shift operators. Here we followed the "principle of the least surprise" to ensure that cross-language programming has to deal with the least possible incompatibilities, and that on a given platform C compilers may at least implement the same model as the C++ compilers.

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 propose to change this in line with the changes in C++. That is to 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

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

adds potential case analysis to programs. C++ removes this feature, and so we propose to do the same.

3.3. Align bit-field specifications with C++

In its new specifications, C++ is more restrictive for bit-fields. In particular since now there is no ambiguity how a signed or unsigned integer type with any width is structured, a bit field of a given width has a unique bit representation.

C++ allows more types as specified bit-field types than C, which only guarantees **`signed`**, **`unsigned`** and **`_Bool`**. But, C also allowed implementations to extend the possible types, so all implementations / platforms / ABIs that want to be compatible with C++ may just add all the types that C++ allows.

The only ambiguity to resolve is to know if a bit field can be wider than the width of the underlying specified type. C++ resolves this by allowing to specify a wider number of bits than in the specified type, but by forcing the excess bits to be padding. C17 forbids excess bits. Since this choice effects the layout of **`struct`** types and since such types with oversized bit-fields would then not be portable from C++ to C, we think that C2x should be extended in that direction. Such a choice is conservative, because it does not invalidate existing code.

4. ALIGN INTEGER OPERATIONS WITH C++

By allowing just two's complement as the sign representation, some specifications for operations on signed integers become simpler, and we try to take advantage of this by removing some "dead specifications" from C2x. Nevertheless there are two operations that need a bit more attention, because C++ is changing the behavior, here.

4.1. Conversion to signed integer types

C++ takes a relatively radical step by now forcing conversion to signed types to be a modulo operation, that is, any integer that is converted to a signed type has exactly the same bit representation as would have a conversion to the corresponding unsigned type.

C's practice seems not to be in accordance with this, and in particular there are implementations that may raise signals when a value is converted that does not fit into the target type. Since this C++ change does not concern interface compatibility between the two languages, we think that it is possible and necessary for C to keep the possibility of raising a signal.

On the other hand, C17 also allows to convert to completely different values than those that are indicated above. We are not aware of any implementation that uses this possibility, and so we propose to cut it off.

In summary, for conversion to signed integer types, we propose that in the case that the signed target type cannot hold the value either

— an implementation defined signal is raised, or
— the value is converted to the same bit representation as for conversion to the corresponding unsigned type.

4.2. Bitwise shift operations

Here also, C++ takes a relatively radical step and forces shift operations on signed integer types to be:

— a modulo operation for left shift, in particular shift across and into the sign bit is legal;
— an arithmetic shift with sign extension for right shift.

The first choice does not concur with current practice in C, where signed overflow via left shift is undefined. Optimizers depend on this property of the left shift operation, and thus we don't think that this can easily be changed.

The second choice, seems easier to take into C, since here we previously only had different implementation defined values that could be the result, but not undefined behavior. Therefore we propose to apply the changes from C++ also to C.

5. OTHER IMPROVEMENTS WITHOUT NORMATIVE CHANGES

5.1. Enumerated types

A comparison with C++ shows that there is now quite a gap in the understanding of enumerated types between both languages. Unifying the different approaches needs (and has already produced) specific papers that address these problems and possible extension for C.

One difference though is apparent that can be fixed easily, which is a complete lack of specification in C how enumerated types are represented and how the conversion between enumerated types and other integer types is to be performed. C provides a notion of a compatible integer type for each enumerated type, but only the context suggests that representation and conversion (and thus use in arithmetic, for example) has to be done according to that compatible integer type.

The present proposal fills this gap by making this rule explicit. This is simply done by adding an "as-if" rules for the representation and conversion. This allows to convert from and to enumerated types as-if the conversion where from or to the corresponding compatible type. This only "works" in C because enumerated types are not observable via **_Generic** expressions, and so the concrete result type for C is not observable.

5.2. Width, minimum and maximum macros for integer types

With the proposed 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 propose to change the presentation to be 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.

6. 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 Clauses 5.2.4.2.1, 6.2.6.2, 6.3.1.3, 6.7.2.1 p9

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**

```
1  BOOL_WIDTH                    1
```

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

```
1  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**

```
1 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**

```
1 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**

```
1 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**

```
1 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

```
1 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 expansion of the macro **CHAR_MIN** shall be the same as for **SCHAR_MIN** and the expansion of macro **CHAR_MAX** as for **SCHAR_MAX**. Otherwise, the expansion of **CHAR_MIN** shall be **0** and the one of **CHAR_MAX** shall be the same as for **UCHAR_MAX**.[3]

[3] See **??**.

...

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 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 An enumerated type shall have the same representation as its corresponding compatible type, see **??**.

5 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.*

...

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than **_Bool**, if any of the two types is an enumerated type, the following rules are applied as for the corresponding compatible type. If the converted value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is signed and the value cannot be represented in it, an implementation-defined signal may be raised.
3 Otherwise, the result is the unique value of the destination type that is congruent to the source integer modulo 2^N, where N is the width of the destination type.
...

6.7.2.1 p9

9 A bit-field declares a member that consists of a number of bits as specified by the constant expression (including a sign bit, if any), called its *width*. If the width exceeds the width of an object of the type that would be specified were the colon and expression omitted, the extra bits are padding bits.[4] That is, a nonzero-width bit-field that is not of type **_Bool** behaves as if having a hypothetical signed or unsigned integer type with a width that is the minimum of the specified constant expression and the width of its specified type.[5] A nonzero-width bit-field of type **_Bool** obeys the same rules for its value representation as other unsigned integer types and has the semantics of a **_Bool**.

[4] The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.
[5] As specified in **??** above, if the actual type specifier used is **int** or a typedef-name defined as **int**, then it is implementation-defined whether the bit-field is signed or unsigned.

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. Also note that it includes the changes for extended integer types that are presented in document **N2303**[6].

[6] http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2303.pdf

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:

— allow extended integer types wider than **`intmax_t`** and **`uintmax_t`**

— remove obsolete sign representations and integer width constraints

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

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 struct-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**, 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
```

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**.

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

```
~~SCHAR_MIN                     -127 // -(2^7 - 1)~~
USHRT_WIDTH                     16
```

— ~~maximum value for an object 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
```

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
```

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**~~

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

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

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

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

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

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

~~maximum value for an object 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**~~

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

4 If an object of type **char** can hold negative values, the ~~value of~~ expansion of the macro **CHAR_MIN** shall be the same as ~~that of~~ for **SCHAR_MIN** and the ~~value of **CHAR_MAX** shall be the same as that of~~ expansion of macro **CHAR_MAX** as for **SCHAR_MAX**. Otherwise, the ~~value~~ expansion of **CHAR_MIN** shall be ~~0 and the value~~ **0** and the one of **CHAR_MAX** shall be the same as ~~that of~~ for **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_559__** shall implement floating point types and arithmetic conforming to IEC 60559 as specified in Annex F. An implementation that defines **__STDC_IEC_559_COMPLEX__** shall implement complex types and arithmetic conforming to IEC 60559 as specified in Annex G.

2 The following parameters are used to define the model for each floating-point type:

s	sign (± 1)
b	base or radix of exponent representation (an integer > 1)
e	exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
p	precision (the number of base-b digits in the significand)
f_k	nonnegative integers less than b (the significand digits)

3 A *floating-point number* (x) is defined by the following model:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{\min} \le e \le e_{\max}$$

4 In addition to normalized floating-point numbers ($f_1 > 0$ if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as *subnormal floating-point numbers* ($x \neq 0$, $e = e_{\min}$, $f_1 = 0$) and *unnormalized floating-point numbers* ($x \neq 0$, $e > e_{\min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring

[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.

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.[50)]

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.[51)] 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.[52)] 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.[53)] 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 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

[50)] 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 $2^{\text{CHAR\_BIT}} - 1$.

[51)] 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.

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

[53)] 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.

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 An enumerated type shall have the same representation as its corresponding compatible type, see 6.7.2.2.

5 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~~

6 **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.

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

8 **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.[54)] 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 two structures, corresponding members shall be declared in the same order. For two structures or

[54)] Two types need not be identical to be compatible.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

— The rank of **char** shall equal the rank of **signed char** and **unsigned char**.

— The rank of **_Bool** shall be less than the rank of all other standard integer types.

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

2 The following may be used in an expression wherever an **int** or **unsigned int** may be used:

— An object or expression with an integer type (other than **int** or **unsigned int**) whose integer conversion rank is less than or equal to the rank of **int** and **unsigned int**.

— A bit-field of type **_Bool**, **int**, **signed int**, or **unsigned int**.

If an **int** can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*.[57] All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a "plain" **char** can hold negative values is implementation-defined.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

6.3.1.2 Boolean type

1 When any scalar value is converted to **_Bool**, the result is 0 if the value compares equal to 0; otherwise, the result is 1.[58]

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than **_Bool**, if any of the two types is an enumerated type, the ~~value can be represented by the new type, it is unchanged.~~

~~Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that~~ following rules are applied as for the corresponding compatible type. If the converted value can be represented ~~in~~ by the new type~~until the value is in the range of the new type.~~ , it is unchanged.

2 ~~Otherwise,~~ Otherwise, if the new type is signed and the value cannot be represented in it~~; either the result is implementation-defined or~~ , an implementation-defined signal ~~is raised.~~ may be raised.

3 Otherwise, the result is the unique value of the destination type that is congruent to the source integer modulo 2^N, where N is the width of the destination type.

[57] The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary `+`, `-` , and `~` operators, and to both operands of the shift operators, as specified by their respective subclauses.

[58] NaNs do not compare equal to 0 and thus convert to 1.

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`.[76)]

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'`.

Forward references: common definitions `<stddef.h>` (7.19), the **`mbtowc`** function (7.22.7.2), Unicode utilities `<uchar.h>` (7.28).

6.4.5 String literals

Syntax

1 *string-literal:*

encoding-prefix$_{\text{opt}}$ " *s-char-sequence*$_{\text{opt}}$ "

[76)] 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).

```
{
      int n = 4, m = 3;
      int a[n][m];
      int (*p)[m] = a;  // p == &a[0]
      p += 1;           // p == &a[1]
      (*p)[2] = 99;     // a[1][2] == 99
      n = p - a;        // n == 1
}
```

11 If array `a` in the above example were declared to be an array of known constant size, and pointer `p` were declared to be a pointer to an array of the same known constant size (pointing to `a`), the results would be the same.

Forward references: array declarators (6.7.6.2), common definitions `<stddef.h>` (7.19).

6.5.7 Bitwise shift operators

Syntax

1 *shift-expression:*
additive-expression
shift-expression **«** *additive-expression*
shift-expression **»** *additive-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The integer promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined.

4 The result of `E1 << E2` is ~~`E1` left-shifted `E2` bit positions; vacated bits are filled with zeros.~~ the unique value congruent to $E1 \times 2^{E2}$ modulo 2^N, where N is the width of the type of the result.[108] ~~If `E1` has an unsigned type, the value of the result is $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type .~~ If `E1` has a signed type and nonnegative value, and $E1 \times 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

5 The result of `E1 >> E2` is ~~`E1` right-shifted `E2` bit positions.~~ the arithmetic quotient $E1/2^{E2}$ with any fractional part discarded.[109] ~~If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of $E1/2^{E2}$. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.~~

6.5.8 Relational operators

Syntax

1 *relational-expression:*
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

Constraints

2 One of the following shall hold:

[108] `E1` is left-shifted `E2` bit positions; vacated bits are filled with zeros.
[109] `E1` is right-shifted `E2` bit positions. Right-shift on signed integer types is an arithmetic right shift, which performs sign-extension.

5 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter.

6 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[112)]

7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

6.5.10 Bitwise AND operator

Syntax

1 *AND-expression:*
equality-expression
AND-expression **&** *equality-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the binary & operator is the bitwise AND of the operands (that is, each bit in the value representation of the result is set if and only if each of the corresponding bits in the value representations of the converted operands is set).

6.5.11 Bitwise exclusive OR operator

Syntax

1 *exclusive-OR-expression:*
AND-expression
exclusive-OR-expression **^** *AND-expression*

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the ^ operator is the bitwise exclusive OR of the operands (that is, each bit in the value representation of the result is set if and only if exactly one of the corresponding bits in the value representations of the converted operands is set).

6.5.12 Bitwise inclusive OR operator

Syntax

1 *inclusive-OR-expression:*
exclusive-OR-expression
inclusive-OR-expression **|** *exclusive-OR-expression*

[112)] Two objects can be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

Constraints

2 Each of the operands shall have integer type.

Semantics

3 The usual arithmetic conversions are performed on the operands.

4 The result of the | operator is the bitwise inclusive OR of the operands (that is, each bit in the value representation of the result is set if and only if at least one of the corresponding bits in the value representations of the converted operands is set).

6.5.13 Logical AND operator

Syntax

1 *logical-AND-expression:*
inclusive-OR-expression
logical-AND-expression **&&** *inclusive-OR-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The && operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.

4 Unlike the bitwise binary & operator, the && operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares equal to 0, the second operand is not evaluated.

6.5.14 Logical OR operator

Syntax

1 *logical-OR-expression:*
logical-AND-expression
logical-OR-expression **||** *logical-AND-expression*

Constraints

2 Each of the operands shall have scalar type.

Semantics

3 The || operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.

4 Unlike the bitwise | operator, the || operator guarantees left-to-right evaluation; if the second operand is evaluated, there is a sequence point between the evaluations of the first and second operands. If the first operand compares unequal to 0, the second operand is not evaluated.

6.5.15 Conditional operator

Syntax

1 *conditional-expression:*
logical-OR-expression
logical-OR-expression **?** *expression* **:** *conditional-expression*

6.7.2.1 Structure and union specifiers

Syntax

1 *struct-or-union-specifier:*
struct-or-union *identifier*$_{\text{opt}}$ **{** *struct-declaration-list* **}**
struct-or-union *identifier*
struct-or-union:
struct
union
struct-declaration-list:
struct-declaration
struct-declaration-list *struct-declaration*
struct-declaration:
specifier-qualifier-list *struct-declarator-list*$_{\text{opt}}$ **;**
static_assert-declaration
specifier-qualifier-list:
type-specifier *specifier-qualifier-list*$_{\text{opt}}$
type-qualifier *specifier-qualifier-list*$_{\text{opt}}$
alignment-specifier *specifier-qualifier-list*$_{\text{opt}}$
struct-declarator-list:
struct-declarator
struct-declarator-list **,** *struct-declarator*
struct-declarator:
declarator
bit-field-specification
bit-field-specification:
declarator$_{\text{opt}}$ **:** *constant-expression*

Constraints

2 A struct-declaration that does not declare an anonymous structure or anonymous union shall contain a struct-declarator-list.

3 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

4 The ~~expression that specifies the width of~~ constant expression in a bit-field specification shall be an integer constant expression with a nonnegative value. ~~that does not exceed the width of an object of the type that would be specified were the colon and expression omitted.~~ If the value is zero, the declaration shall have no declarator.

~~A~~ A declaration with a bit-field specifier shall have a type that is a qualified or unqualified version of **_Bool**, **signed int**, **unsigned int**, or some other implementation-defined type. It is implementation-defined whether atomic types are permitted.

Semantics

5 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

6 Structure and union specifiers have the same form. The keywords **struct** and **union** indicate that the type being specified is, respectively, a structure type or a union type.

7 The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. The struct-declaration-list is a sequence of declarations for the members of the structure or union. If the struct-declaration-list does not contain any named members, either directly or via an anonymous structure or anonymous union, the behavior is undefined. The type is

incomplete until immediately after the } that terminates the list, and complete thereafter.

8 A member of a structure or union may have any complete object type other than a variably modified type.[124] In addition, a member may be declared to consist of a specified

9 A bit-field declares a member that consists of a number of bits as specified by the constant expression (including a sign bit, if any). Such a member is called a ;[125] its width is preceded by a colon , called its *width*.

A If the width exceeds the width of an object of the type that would be specified were the colon and expression omitted, the extra bits are padding bits.[125] That is, a nonzero-width bit-field is interpreted as having a that is not of type **_Bool** behaves as if having a hypothetical signed or unsigned integer type consisting with a width that is the minimum of the specified number of bitsconstant expression and the width of its specified type.If the value 0 or 1 is stored into a[126] A nonzero-width bit-field of type **_Bool** , the value of the bit-field shall compare equal to the value stored; a **_Bool** bit-field obeys the same rules for its value representation as other unsigned integer types and has the semantics of a **_Bool**.

10 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

11 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[127] As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

12 An unnamed member whose type specifier is a structure specifier with no tag is called an *anonymous structure*; an unnamed member whose type specifier is a union specifier with no tag is called an *anonymous union*. The members of an anonymous structure or union are considered to be members of the containing structure or union, keeping their structure or union layout. This applies recursively if the containing structure or union is also anonymous.

13 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

14 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.

15 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

16 There may be unnamed padding at the end of a structure or union.

17 As a special case, the last member of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were

[124] A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

[125] The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

[125] The unary & (address-of) operator cannot be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

[126] As specified in 6.7.2 above, if the actual type specifier used is **int** or a typedef-name defined as **int**, then it is implementation-defined whether the bit-field is signed or unsigned.

[127] An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

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.

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.[264] 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,[265] <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").

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.

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:

[264] See "future library directions" (7.31.10).
[265] Some of these types might denote implementation-defined extended integer types.

3 **NOTE 2** It follows from the definitions that greatest-width integer types are at least 64 bit wide.

4 **NOTE 3** Extended integer types that are not referred by the above list and that have values not representable by **signed long long int** or **unsigned long long int**, respectively, need not be representable by the greated-width types.

Recommended practice

5 Unless some **typedef** in the library clause enforces otherwise, it is recommended to resolve these types to **signed long int** or **signed long long int** and the corresponding unsigned counterpart. It is recommended that the same set of integer literals is consistently accepted by all compilation phases, even if greatest-width types are chosen that are wider than **signed long long int**.

7.20.2 Characteristics 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. The maximum and possibly minimum macros are defined as by 5.2.4.2.

2 Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives~~, and 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~~ An implementation shall define only the macros corresponding to those typedef names it actually provides.~~maximum values~~ [267)]

7.20.2.1 Width of exact-width integer types

1 width of exact-width ~~unsigned integer types~~ integer types

~~UINT*N*_MAX~~	~~exactly~~ $2^N - 1$
INT*N*_WIDTH	exactly N
UINT*N*_WIDTH	exactly N

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

7.20.2.2 Width of minimum-width integer types

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

~~UINT_LEAST*N*_MAX~~	$2^N - 1$
INT_LEAST*N*_WIDTH	exactly UINT_LEAST*N*_WIDTH
UINT_LEAST*N*_WIDTH	N

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

7.20.2.3 Width of fastest minimum-width integer types

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

~~UINT_FAST*N*_MAX~~	$2^N - 1$
INT_FAST*N*_WIDTH	exactly UINT_FAST*N*_WIDTH
UINT_FAST*N*_WIDTH	N

~~minimum value~~

7.20.2.4 Width of integer types capable of holding object pointers

1 width of pointer-holding ~~signed integer type~~ integer types

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

~~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~~

~~minimum value~~

7.20.2.5 Width of greatest-width integer types

1 width of greatest-width ~~signed integer type~~ integer types

~~INTMAX_MIN~~	$-(2^{63}-1)$
INTMAX_MAX	exactly UINTMAX_WIDTH
UINTMAX_MAX	64

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

7.20.3 Characteristics 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. The maximum and minimum macros are defined as by 5.2.4.2. If the type is unsigned a minimum macro is defined to expand to 0.

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 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.[268)]

— ~~limits of~~

— ~~limits of~~ width of **ptrdiff_t**

PTRDIFF_WIDTH	16

— ~~limit of~~ width of **sig_atomic_t**

SIG_ATOMIC_WIDTH	8

— ~~limits of~~ width of **size_t**

SIZE_MAX	16

— ~~limits of~~ width of **wchar_t**

WCHAR_WIDTH	8

~~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.~~

— width of **wint_t**

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

WINT_WIDTH	16

~~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.~~

~~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)
```

7.29 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.[330)]

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);[331)] 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 ~~??~~7.20.3); 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.[332)] 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.[333)]

[330)] See "future library directions" (7.31.16).
[331)] **wchar_t** and **wint_t** can be the same integer type.
[332)] The value of the macro **WEOF** can differ from that of **EOF** and need not be negative.
[333)] 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.[361)]

```
      #define BOOL_WIDTH                      1
      #define CHAR_BIT                        8
      #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 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 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 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
      #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 USHRT_WIDTH                    16
      #define UINT_MAX                   65535  // 2^UINT_WIDTH − 1
      #define UINT_WIDTH                     16
      #define ULONG_MAX               4294967295  // 2^ULONG_WIDTH − 1
      #define ULONG_WIDTH                    32
      #define ULLONG_MAX    18446744073709551615  // 2^ULLONG_WIDTH − 1
      #define ULLONG_WIDTH                   64
```

[361)] The minimum value of a signed integer type is not expressible in the same type when using a decimal literal without suffix and a sign. 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.

— 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~~ possible 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.3.6 Floating point

1 — The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

— The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` (5.2.4.2.2).

— The rounding behaviors characterized by non-standard values of **FLT_ROUNDS** (5.2.4.2.2).

— The evaluation methods characterized by non-standard negative values of **FLT_EVAL_METHOD** (5.2.4.2.2).

— The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

— The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).