intmax

$_t$, *awayout*

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N2425**

v 2

September 13, 2019

intmax_t, a way out v.2

Ease the definition of extended integer types

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

The specifications of types `[u]intmax_t` and extended integer types lack to provide the extensibility feature for which they are designed. As a consequence existing "64 bit" implementations are not able to add standard conforming interfaces to their 128 bit or 256 bit integer types without breaking ABI compatibility.

Changelog:

— Changes from v.1, n2303:
  — A detailed list of the types that are covered by `[u]intmax_t` is added to the normative text.
  — A new pair of types is added that also captures extended integer types and that can be used for `printf` and similar, but that should not be part of any API or ABI.
  — The co-existence of "least" and "fast" signed and unsigned pairs of is enforced.
  — Range inclusion of "least" and "fast" types with width $N < M$ is enforced.
  — Introduce type-generic macros `int_abs`, `int_div`, `int_max`, and `int_min`, intended to replace and complement `imaxabs` and `imaxdiv` functions.
  — Substitute integer-power functions `compoundn`, `pown` and `rootn` by type-generic macros of the same name that avoid the use of the `[u]intmax_t` types.
  — Substitute integer-power functions `fromfp` and similar by type-generic macros `toint`, `touint`, `tointx` and `touintx` that avoid the use of the `[u]intmax_t` types.
— A previous version of this proposal has been discussed in message SC22WG14.15569 and the depending thread on the WG14 reflector.

Note: This paper assumes the prior integration of N2412, "*Two's complement sign representation for C2x*"

Contents

1. PROBLEM DESCRIPTION

The interaction between the definition of extended integer types and `[u]intmax_t` has resulted in a lack of extensibility for existing ABI. Platforms that fixed their specifications for the basic integer types and for `[u]intmax_t` cannot add an extended integer type that is wider than their current `[u]intmax_t` to their specification. As the current text of the C standard stands, such an addition would force a redefintion of `[u]intmax_t` to the wider types. This would have the following consequences:

— The parts of the C library that use `[u]intmax_t` (specific functions but also `printf` and friends) must be rewritten or recompiled with the new ABI and become binary incompatible with existing programs.
— Programs compiled with the new ABI would be binary incompatible on platforms that have not been upgraded.
— The preprocessor of the implementation must be re-engineered to comply to the standard. In particular, there would occur severe specification problems for preprocessor numbers and their evaluation. *E.g* the *value* of `ULLONG_MAX+1` is not expressible as a literal in the language proper but would be for the preprocessor. The *expression* `ULLONG_MAX+1` would evaluate to `true` in a preprocessor conditional but to `0` (`false`) in later compilation phases.

As a consequence of these difficulties the concept of "extended integer type" is merely unused by implementations. I have not heard of any implementation that uses this concept. So as it stands this idea of "extended integer type" is basically a failure, nobody uses it, and `intmax_t` is usually just `long` or `long long`.

This has lead to a sensible backlog for platforms such as `gcc` or `clang` that provide emulated 128 bit integer types (`__[u]int128_t`) on 64 bit platforms. They are not able to provide them as "extended integer types" in the sense of the C standard. More and more processor platforms even provide rudimentary support of 128 or 256 bit integers in hardware (*e.g* Intel's `AVX` vector unit), so it would really be productive to give more slack to implementations to integrate these types into existing ABI.

Generally, we should not block implementations that are able to provide exact-width integer types for $N > 64$. These types can for example be used efficiently for bitsets, UUIDs, cryptography, checksums or networking (ipv6). They have well-defined standard interfaces in the form of (`[u]int`N`_t`) with easy to use feature tests.

2. SUGGESTED CHANGES

2.1. Changes directly concerning `[u]intmax_t`

I suggest to change the specification of `[u]intmax_t` such that they are only at least as wide as any integer type *used* by the standard. Thereby the greatest-width types do not have to cover *all* integer types, in particular not extended integer types that might be added later to an ABI.

The change to the standard can be isolated in 7.20.1.15. The two main changes would be changing p1 and add a recommended practice section:

> ### 7.20.1.5 Greatest-width integer types
>
> The types `intmax_t` and `uintmax_t` designate a signed and an unsigned integer type, respectively, that are at least as wide as any basic integer type, the types
>
> | | | | |
> |---|---|---|---|
> | `char16_t` | `int_least64_t` | `size_t` | `wchar_t` |
> | `char32_t` | `ptrdiff_t` | `uint_fast64_t` | `wint_t` |
> | `int_fast64_t` | `sig_atomic_t` | `uint_least64_t` | |

> and, provided they exist, **intptr_t** and **uintptr_t**. These types are required.
>
> **Note:** Extended integer types that are not referred by the above list and that are wider than **signed long long int** may also be wider than **intmax_t**.
>
> **Recommended practice:** Unless some **typedef** in the library clause enforces otherwise, it is recommended to resolve these types to **long** or **long long** 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 **long long**.

QUESTION 1. *Shall the requirements for the types* **[u]intmax_t** *be relaxed to cover only basic integer types and other semantic integer types as proposed in N2425?*

2.2. Marginaly corrections for exact width macros and similar

Relaxing the requirements on **[u]intmax_t** and have extended integer types that are wider that that, has one marginal implication: wide extended integer types might now have literals that cannot be use as preprocessing numbers within **#if** expressions. We propose to force a diagnosis of such situations (this all happens in the preprocessor) by formulating constraints that forbid such a use.
For the **_C** macros, the text for that is an addition to the end of 7.20.4 p1:

> For types wider than **uintmax_t**, the macros shall only be defined if the implementation provides integer literals for the type that are suitable to be used in **#if** preprocessing directives. Otherwise, the definition of these macros is mandatory for any of the types that are provided by the implementation.

QUESTION 2. *Shall the* **_C** *macros for minimum-width types that are wider than* **UINTMAX_WIDTH** *be constrained as proposed in N2425?*

For the **_MAX** and **_MIN** macros, the text has a similar addition to the end of 7.20.5 p1:

> For types wider than **uintmax_t** and for which the corresponding minimum width integer constant macro with suffix **_C** is not defined, 7.20.4.1 and 7.20.4.2, the macros are not necessarily suitable to be used in **#if** preprocessing directives.

and to the addition of *Constraints* and *Recommended practice* as follows:

> **Constraints**
>
> If N is greater than **UINTMAX_WIDTH** and the corresponding macros **INT** N**_C** or **UINT** N**_C** are not defined, the derived **_MIN** and **_MAX** macros for the exact-width, minimum-width and fastest minimum-width types for N shall not be used in an **#if** preprocessing directive, unless they are operand of the **defined** operator.[FNT] [FNT] This constraint reflects the fact that these macros may have numerical values that exceed the largest value that is representable during preprocessing. In that case these constants will generally be expressed by constant expressions that are more complex and not suitable for preprocessing.
>
> **Recommended practice:** Because of the above constraints, applications should prefer the **_WIDTH** macros over the **_MIN** or **_MAX** macros for feature tests in **#if** preprocessing directives.

QUESTION 3. *Shall* `_MAX` *and* `_MIN` *macros of the exact-with and least and fasted minimum-width types that are wider than* `UINTMAX_WIDTH` *be constrained not to be used during preprocessing as proposed in N2425?*

2.3. Tighten the rules for least and fast minimum-width integer types

When trying to formulate the features proposed in this paper, we noticed some lack of precisions and requirements for the least and minimum-width integers. For example no text currently explicitly says that the pairs of such designated types have to be the corresponding signed and unsigned integer types, nor is the existence of these types enforced, if the fixed-width types exist.

As such I don't think they are a big deal and everybody does this probably in a reasonable way. Nevertheless, I think it would help if these aspects were clarified. The corresponding text is relatively straight forward and can be found in the annex.

QUESTION 4. *Shall complementarity of signed and unsigned least and fast minimum-width type pairs be enforced as proposed in N2425?*

QUESTION 5. *Shall the least and fast minimum-width type pairs be required if the corresponding exact-width type is provided as proposed in N2425?*

QUESTION 6. *Shall the value ranges for least and fast minimum-width types with width $N < M$ be enforced as proposed in N2425?*

2.4. New type aliases for the widest type pair

A new second paragraph should be added to 7.20.1.5 to introduce the new types. We only would require them to be at least as wide as all integer types that are introduced in `<stdint.h>`; that is they would be at least as wide as `[u]intmax_t` and all other optional exact-width and least and fasted minimum-width types that are provided by the plate-forme.

> The types `int_leastmax_t` and `uint_leastmax_t` designate a signed and an unsigned integer type, respectively, that are at least as wide as any integer type defined by the header `<stdint.h>`. These types are required.

The note introduced previously is then amended and results in three notes and an example as follows.

> **Note 1:** The `intmax_t` and `uintmax_t` types are intended to provide a fallback for applications that deal with integers for which they lack specific type information. This mainly occurs for two different situations. First, for integers that appear in conditional inclusion (`#if` expressions, 6.10.1) they provide fallback types that capture the implementation specific capabilities during translation phase 4. Second, for some semantic type definitions that resolve to implementation specific types there are no special provisions for `printf`, `scanf` or similar functions. In particular, the `intmax_t` and `uintmax_t` types are intended to represent values of all types listed above and also the exact-width integer types for all $N \leq 64$.
>
> **Note 2:** Extended integer types that are not referred by the above list and that are wider than `signed long long int` may also be wider than `intmax_t`. The types `intmax_t` and `int_leastmax_t` may then be different.
>
> **Note 3:** The `int_leastmax_t` and `uint_leastmax_t` types are intended to provide a fallback for applications that deal with unknown integer types that are potentially wider than `intmax_t` or `uintmax_t`.

Example: An implementation that has historically fixed its type **intmax_t** to a 64 bit type, and seeks to add a 128 bit integer exact-width type to its extended integer types, may do so by providing types uint128_t, uint_least128_t, uint_least128_t, uint_leastmax_t and the corresponding signed types and macros of stdint.h and inttypes.h (7.8.1) without breaking binary compatibility.

Application code can then query the type and print it by using the appropriate macros:

```
1   #include <stdint.h>
2   #include <stdio.h>
3   #include <inttypes.h>
4   #ifdef UINT128_MAX          // ok, because #ifdef
5   typedef uint128_t bitset;
6   #else
7   typedef uint_least64_t bitset;
8   #endif
9   int main(void) {
10          bitset all = -1;
11          printf("the largest set is %#"
                PRIXLEASTMAX "\n", (uint_leastmax_t)
                all);
12  }
```

The "recommended practice" introduced previously is then amended to make clear that [u]int_leastmax_t should never take part in any API or ABI.

> Implementations and applications should not use the types int_leastmax_t and uint_leastmax_t to describe application programmable interfaces.

To be fully operational, also some macros (**_MAX** etc) must be added to the text for stdint.h. These additions are straight forward and can be seen in the Appendix.
The "recommended practice" introduced previously is then amended to make clear that these macros should be used with care during preprocessing.

> If the macros INT_LEASTMAX_C and UINT_LEASTMAX_C are not defined, the derived macros INT_LEASTMAX_MIN, INT_LEASTMAX_MAX and UINT_LEASTMAX_MAX shall not be used in an **#if** preprocessing directive, unless they are operand of the **defined** operator.

QUESTION 7. *Shall types* [u]int_leastmax_t *be added to the C standard as proposed in N2425?*

QUESTION 8. *If not, shall types with the indicated semantics of* [u]int_leastmax_t *as proposed in N2425 but with different names be added to the C standard?*

2.5. Chasing [u]intmax_t from standard interfaces

To avoid such situations where implementations get stuck because of early ABI choices, I think that it would be good to phase out all interfaces that use **[u]intmax_t**, and to replace them by type-generic macros, instead. These interfaces in the C standard are

imaxabs	**strtoumax**	**compoundn**	fromfp	ufromfp
imaxdiv	**wcstoimax**	**pown**	fromfpx	ufromfpx
strtoimax	**wcstoumax**	**rootn**		

Only the first six appear already in a published version of the standard, so we should not remove them, just declare them obsolescent. The others are current additions that are not yet published, so it is still time to completely replace them by the alternatives as proposed below.
The basic idea for all the replacements is to use type-generic interfaces.

— For the first six, standard functions already exist for the wide basic integer types. Since [u]**intmax_t** have never been used in the field with other types than **long** or **long long**, this is just code duplication and the **long long** interface could clearly have worked all along.
— In contrast to a catch-all solution with [u]**intmax_t** a type-generic solution always chooses the right interface for the type at hand and avoids useless conversion to (for the function argument) and from (for the return value) a wider type.
— Just specifying type-generic macros and not specific functions, allows implementations to provide the functionally by functions that only have internal names or other mechanisms to their liking. This reduces the risk of name conflicts in the source (only may occur when the header is included) and during linking.
— Type-generic macros are also more flexible, because implementations may add cases as they go, *e.g* if they introduce 128 bit floating types and integer types at the same time.

The text for all these type-generic interfaces are just additions and clearly identifiable in the appendix, so it is not repeated here. We explain them more in detail in the following sections.
For the {str|**wcs**}**to**[ui]max functions, the proposal is even simpler. The underlying family of functions cannot use the type of their argument to distinguish which type should be returned. If we want to phase out the [u]**intmax_t** types, such functions should simply not be used, but the appropriate function for the sought user type should be used directly.

QUESTION 9. *Shall we mark the* {str|**wcs**}**to**[ui]max *functions as obsolescent as proposed in N2425?*

2.5.1. Type-generic macros for common integer operations

— The interface **imaxabs** has the particularity that it is not defined for all inputs, because the mathematical value **-INTMAX_MAX** is out of range of the type **intmax_t**. This is an unnecessary restriction because:
  — We know that the result is not negative, anyhow.
  — The type **uintmax_t** comprises all the possible return values.
  Therefore we propose a type-generic macro that does not follow the previous interface where we have the same return type as the parameter type. Instead, it returns the unsigned type of the parameter type. Thereby the proposed type-generic solution is well-defined for all argument values.

QUESTION 10. *Shall we introduce the type-generic macro* int_abs *to the C standard as proposed in N2425?*

QUESTION 11. *Shall we mark the* **imaxabs** *function as obsolescent as proposed in N2425?*

— The interface **imaxdiv** even introduces a new type **imaxdiv_t** that is unnecessary because for all current implementations **lldiv_t** would do. Also the current version with a function has no defined behavior if the function is called with an unsigned argument that is larger

than **INTMAX_MAX**. With our proposal it is a constraint violation to call the type-generic macro with a first unsigned and second signed argument where there is no signed supertype of the first argument type.

QUESTION 12. *Shall we introduce the type-generic macro* `int_div` *to the C standard as proposed in N2425?*

QUESTION 13. *Shall we mark the* **imaxdiv** *function as obsolescent as proposed in N2425?*

— As a little sidetrack we also propose to complete the picture by adding maximum and minimum type-generic macros to the same clause. The first can profit from a similar observation as `int_abs`. In case that the argument types are mixed signed and unsigned, it is sufficient to return an unsigned type to cover all the possible return values.

QUESTION 14. *Shall we introduce the type-generic macro* `int_max` *to the C standard as proposed in N2425?*

— For the minimum, things are a bit more complicated. Here we have to require that a common signed supertype for the two argument types must exist.

QUESTION 15. *Shall we introduce the type-generic macro* `int_min` *to the C standard as proposed in N2425?*

2.5.2. Integer-power type-generic macros

The current version of C2x proposes to add three new "integer-power function" of different flavor. In the current proposal the type of the integer parameter is **intmax_t**. This is not strictly necessary, **long long** should be largely enough. Our proposal uses an unspecified integer type that could vary between implementations, as long as it is wide enough to cover the possible values for the operation.

QUESTION 16. *Shall we replace the functions of type* **compoundn**, **rootn** *and* **pown** *by the type-generic macros as proposed in N2425?*

2.5.3. Nearest integer type-generic macros

The newly introduced nearest integer type functions `fromfp`, `fromfpx`, `ufromfp`, and `ufromfpx` all use a return type of **intmax_t**, but for no compelling reason. As already observed for other functions, a return type of **long long** would generally be sufficient, to capture the possible return values.
A type-generic macro has the advantage that the return type has not to be fixed once and for all, but that the addition of new floating point types to an implementation, would allow to define an appropriate return type that is adapted to the situation.
Since some implementations that implement the previous API (and depending ABI) might already be out there, we propose to rename the feature such that it better fits into the established naming scheme.

QUESTION 17. *Shall we introduce the type-generic macros* **to**`{u}?`**int**`{x}?` *to the C standard as proposed in N2425?*

QUESTION 18. *Shall we remove the* `{u}?fromfp{x}?` *functions from the C standard as proposed in N2425?*

3. IMPACT

3.1. Existing implemenations and code

With such a change of the C standard, no existing ABI would have to change, and the preprocessor support for integer expressions could remain unchanged.

Since the concept of extended integer types is basically not yet used by implementations, there would also be no impact on the existing code base on existing implementations, even if they chose to extend their ABI by some wider integer types.

3.2. Extensibility of ABI's

This change allows platforms to add specifications of extended integer types more easily. In particular 128 or 256 bit types can be added to 64 bit ABI as long as a conforming naming scheme is chosen. Many implementations do so already in various forms and with non-uniform syntax. With this change they could just **typedef** their extented type to `uint128_t`, say, and provide the corresponding macros `UINT128_MAX`, `UINT128_C`, `PRId128` etc.
There is no need to extend the language to describe additional integer types (such as **long long long**), to add new number literals (`-1ULLL`) or to add **printf** conversion characters for these in the C standard. The use of implementation specific names (`__int128` or `__int128_t`) and implementation specific format specifiers (`"%Q"`) is largely sufficient if appropriately mapped by `<stdint.h>` **typedef** and macros.

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

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

— 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.

7.8 Format conversion of integer types <inttypes.h>

1 The header <inttypes.h> includes the header <stdint.h> and extends it with additional facilities provided by hosted implementations.

2 It declares functions for manipulating greatest-width integers and converting numeric character strings to greatest-width integers, and it declares the obsolescent type

```
imaxdiv_t
```

which is a structure type that is the type of the value returned by the obsolescent **imaxdiv** function. For each type declared in <stdint.h>, it defines corresponding macros for conversion specifiers for use with the formatted input/output functions.[231)]

Forward references: integer types <stdint.h> (7.20), formatted input/output functions (7.21.6), formatted wide character input/output functions (7.29.2).

7.8.1 Macros for format specifiers

1 Each of the following object-like macros expands to a character string literal containing a conversion specifier, possibly modified by a length modifier, suitable for use within the format argument of a formatted input/output function when converting the corresponding integer type. These macro names have the general form of **PRI** (character string literals for the **fprintf** and **fwprintf** family) or **SCN** (character string literals for the **fscanf** and **fwscanf** family),[232)] followed by the conversion specifier, followed by a name corresponding to a similar type name in 7.20.1. In these names, *N* represents the width of the type as described in 7.20.1. For example, **PRIdFAST32** can be used in a format string to print the value of an integer of type **int_fast32_t**.

2 The **fprintf** macros for signed integers are:

~~llll~~	**PRId***N*	**PRIdLEAST***N*	**PRIdFAST***N*	**PRIdMAX**	**PRIdLEASTMAX**	**PRIdPTR**
	PRIi*N*	**PRIiLEAST***N*	**PRIiFAST***N*	**PRIiMAX**	**PRIiLEASTMAX**	**PRIiPTR**

3 The **fprintf** macros for unsigned integers are:

	PRIo*N*	**PRIoLEAST***N*	**PRIoFAST***N*	**PRIoMAX**	**PRIoLEASTMAX**	**PRIoPTR**
~~llll~~	**PRIu***N*	**PRIuLEAST***N*	**PRIuFAST***N*	**PRIuMAX**	**PRIuLEASTMAX**	**PRIuPTR**
	PRIx*N*	**PRIxLEAST***N*	**PRIxFAST***N*	**PRIxMAX**	**PRIxLEASTMAX**	**PRIxPTR**
	PRIX*N*	**PRIXLEAST***N*	**PRIXFAST***N*	**PRIXMAX**	**PRIXLEASTMAX**	**PRIXPTR**

4 The **fscanf** macros for signed integers are:

~~llll~~	**SCNd***N*	**SCNdLEAST***N*	**SCNdFAST***N*	**SCNdMAX**	**SCNdLEASTMAX**	**SCNdPTR**
	SCNi*N*	**SCNiLEAST***N*	**SCNiFAST***N*	**SCNiMAX**	**SCNiLEASTMAX**	**SCNiPTR**

5 The **fscanf** macros for unsigned integers are:

	SCNo*N*	**SCNoLEAST***N*	**SCNoFAST***N*	**SCNoMAX**	**SCNoLEASTMAX**	**SCNoPTR**
~~llll~~	**SCNu***N*	**SCNuLEAST***N*	**SCNuFAST***N*	**SCNuMAX**	**SCNuLEASTMAX**	**SCNuPTR**
	SCNx*N*	**SCNxLEAST***N*	**SCNxFAST***N*	**SCNxMAX**	**SCNxLEASTMAX**	**SCNxPTR**

6 For each type that the implementation provides in <stdint.h>, the corresponding **fprintf** macros shall be defined and the corresponding **fscanf** macros shall be defined unless the implementation does not have a suitable **fscanf** length modifier for the type.

7 **EXAMPLE**

```
#include <inttypes.h>
#include <wchar.h>
int main(void)
{
      uintmax_t i = UINTMAX_MAX;     // this type always exists
      wprintf(L"The largest integer value is %020"
```

[231)] See "future library directions" (7.31.6).

[232)] Separate macros are given for use with **fprintf** and **fscanf** functions because, in the general case, different format specifiers might be required for **fprintf** and **fscanf**, even when the type is the same.

```
                PRIxMAX "\n", i);
        wprintf(L"The largest preprocessor integer value is %#" PRIxMAX "\n", i);
        uint_leastmax_t j = UINT_LEASTMAX_MAX;    // this type always exists
        wprintf(L"The largest extended integer value is %#" PRIxLEASTMAX "\n", j);
        return 0;
    }
```

7.8.2 Functions for greatest-width integer types

1 The functions presented in this subclause are obsolescent features that should not be used in new code. The functions **imaxabs** and **imaxdiv** can be replaced by the type-generic macros **int_abs** and **int_div**, respectively, as introduced below, and **strtoimax**, **strtoumax**, **wcstoimax** and **wcstoumax** can usually be replaced by their counterparts for **long long** types.

7.8.2.1 The imaxabs function

Synopsis

1
```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j);
[[deprecated]] intmax_t imaxabs(intmax_t j);
```

Description

2 The **imaxabs** function computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.[233)]

Returns

3 The **imaxabs** function returns the absolute value.

7.8.2.2 The imaxdiv function

Synopsis

1
```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
[[deprecated]] imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Description

2 The **imaxdiv** function computes `numer / denom` and `numer % denom` in a single operation.

Returns

3 The **imaxdiv** function returns a structure of type **imaxdiv_t** comprising both the quotient and the remainder. The structure shall contain (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type **intmax_t**. If either part of the result cannot be represented, the behavior is undefined.

7.8.2.3 The strtoimax and strtoumax functions

Synopsis

1
```
#include <inttypes.h>
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);
[[deprecated]] intmax_t strtoimax(const char * restrict nptr,
        char ** restrict endptr, int base);
[[deprecated]] uintmax_t strtoumax(const char * restrict nptr,
        char ** restrict endptr, int base);
```

[233)] The absolute value of the most negative number may not be representable.

Description

2 The **strtoimax** and **strtoumax** functions are equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions, except that the initial portion of the string is converted to **intmax_t** and **uintmax_t** representation, respectively.

Returns

3 The **strtoimax** and **strtoumax** functions return the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references: the **strtol**, **strtoll**, **strtoul**, and **strtoull** functions (7.22.1.7).

7.8.2.4 The wcstoimax and wcstoumax functions

Synopsis

1
```
#include <stddef.h>            // for wchar_t
#include <inttypes.h>
intmax_t wcstoimax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
uintmax_t wcstoumax(const wchar_t *restrict nptr, wchar_t **restrict endptr, int base);
[[deprecated]] intmax_t wcstoimax(const wchar_t *restrict nptr,
                wchar_t **restrict endptr, int base);
[[deprecated]] uintmax_t wcstoumax(const wchar_t *restrict nptr,
                wchar_t **restrict endptr, int base);
```

Description

2 The **wcstoimax** and **wcstoumax** functions are equivalent to the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions except that the initial portion of the wide string is converted to **intmax_t** and **uintmax_t** representation, respectively.

Returns

3 The **wcstoimax** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **INTMAX_MAX**, **INTMAX_MIN**, or **UINTMAX_MAX** is returned (according to the return type and sign of the value, if any), and the value of the macro **ERANGE** is stored in **errno**.

Forward references: the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** functions (7.29.4.1.3).

7.8.3 Type-generic macros for common integer operations

1 This clause presents type-generic macros that shall behave as if a function with the indicated prototype were called. The types T (and S were applicable) shall be any supported integer type with a conversion rank of **int** or higher. When called, the corresponding prototype to be applied is determined by the promoted type of the first macro argument for T, and, if applicable by the promoted type of the second argument for S. The return type R is determined as indicated in each clause.

Constraints

2 Arguments to these macros shall be integer expressions.

7.8.3.1 The int_abs type-generic macro

Synopsis

1
```
#include <inttypes.h>
R int_abs(T j);
```

Description

2 The **int_abs** type-generic macro computes the absolute value of an integer j. R is the unsigned

version of T.[234)]

Returns

3 The **int_abs** type-generic macro returns the absolute value.

7.8.3.2 The int_max type-generic macro

Synopsis

1
```
#include <inttypes.h>
R int_max(T j, S k);
```

Description

2 The **int_max** type-generic macro computes the maximum value of two integers j and k. R shall be the common real type of integer types T and S as determined by the usual arithmetic conversions.[235)]

Returns

3 The **int_max** type-generic macro returns the maximum value of its arguments.

7.8.3.3 The int_min type-generic macro

Synopsis

1
```
#include <inttypes.h>
R int_min(T j, S k);
```

Constraints

2 The types T and S shall be integer types such that there exists an integer type that comprises the value ranges of both.[236)]

Description

3 The **int_min** type-generic macro computes the minimum value of two integers j and k. If T and S are either both signed or both unsigned, the return type R shall be their common real type as determined by the usual arithmetic conversions. Otherwise, R shall be the signed integer type of minimum rank that comprises the value ranges of both.

Returns

4 The **int_min** type-generic macro returns the minimum value of its arguments.

7.8.3.4 The int_div type-generic macro

Synopsis

1
```
#include <inttypes.h>
R int_div(T numer, S denom);
```

Constraints

2 If T is an unsigned type and S is a signed type there shall be a signed integer type that comprises the value range of T.[237)]

Description

3 The **int_div** type-generic macro computes `numer / denom` and `numer % denom` in a single operation. If T is an unsigned type and S is a signed type let Q be the signed integer type of minimum rank that comprises the value range of T; otherwise let Q be T. If Q is one of the types **int**, **long int** or **long long int**, R is **div_t**, **ldiv_t** or **lldiv_t**, respectively; otherwise

[234)] If T is a signed type, even the negative of the minimum value of T fits into R.

[235)] If one of T and S is a signed type and the other is unsigned, the maximum of the result is never negative and is thus within the value range of R.

[236)] If T and S are the widest signed and unsigned integer types, respectively, such a type does not exist.

[237)] If T is the widest unsigned integer type such a type does not exist.

`R` is a structure type that contains (in either order) the members `quot` (the quotient) and `rem` (the remainder), each of which has type `Q`.

4 If `S` is signed and `int_abs(numer)< int_abs(denom)` holds, the result has a quotient of value 0 and a remainder of value `numer`. Otherwise, `numer` and `denom` are converted to `Q` before the operation.

Returns

5 The **`int_div`** type-generic macro returns a structure of a type `R` comprising both the quotient and the remainder. If either part of the result cannot be represented, the behavior is undefined.

14 The *math rounding direction macros*

```
FP_INT_UPWARD
FP_INT_DOWNWARD
FP_INT_TOWARDZERO
FP_INT_TONEARESTFROMZERO
FP_INT_TONEAREST
```

represent the rounding directions of the functions **ceil**, **floor**, **trunc**, **round**, and **roundeven**, respectively, that convert to integral values in floating-point formats. They expand to integer constant expressions with distinct values suitable for use as the second argument to the ~~fromfp, ufromfp, fromfpx, and ufromfpx functions~~**toint**, **touint**, **tointx**, and **touintx** type-generic macros, see 7.25.

15 The macro

```
FP_FAST_FMA
```

is optionally defined. If defined, it indicates that the **fma** function generally executes about as fast as, or faster than, a multiply and an add of **double** operands.[246] The macros

```
FP_FAST_FMAF
FP_FAST_FMAL
```

are, respectively, **float** and **long double** analogs of **FP_FAST_FMA**. If defined, these macros expand to the integer constant 1.

16 The macros

```
FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128
```

are, respectively, **_Decimal32**, **_Decimal64**, and **_Decimal128** analogs of **FP_FAST_FMA**.

17 Each of the macros

FP_FAST_FADD	**FP_FAST_DSUBL**	**FP_FAST_FDIVL**	**FP_FAST_FFMA**
FP_FAST_FADDL	**FP_FAST_FMUL**	**FP_FAST_DDIVL**	**FP_FAST_FFMAL**
FP_FAST_DADDL	**FP_FAST_FMULL**	**FP_FAST_FSQRT**	**FP_FAST_DFMAL**
FP_FAST_FSUB	**FP_FAST_DMULL**	**FP_FAST_FSQRTL**	
FP_FAST_FSUBL	**FP_FAST_FDIV**	**FP_FAST_DSQRTL**	

is optionally defined. If defined, it indicates that the corresponding function generally executes about as fast, or faster, than the corresponding operation or function of the argument type with result type the same as the argument type followed by conversion to the narrower type. For **FP_FAST_FFMA**, **FP_FAST_FFMAL**, and **FP_FAST_DFMAL**, the comparison is to a call to **fma** or **fmal** followed by a conversion, not to separate multiply, add, and conversion. If defined, these macros expand to the integer constant **1**.

18 The macros

FP_FAST_D32ADDD64	**FP_FAST_D32MULD64**	**FP_FAST_D32FMAD64**
FP_FAST_D32ADDD128	**FP_FAST_D32MULD128**	**FP_FAST_D32FMAD128**
FP_FAST_D64ADDD128	**FP_FAST_D64MULD128**	**FP_FAST_D64FMAD128**
FP_FAST_D32SUBD64	**FP_FAST_D32DIVD64**	**FP_FAST_D32SQRTD64**
FP_FAST_D32SUBD128	**FP_FAST_D32DIVD128**	**FP_FAST_D32SQRTD128**
FP_FAST_D64SUBD128	**FP_FAST_D64DIVD128**	**FP_FAST_D64SQRTD128**

[246] Typically, the **FP_FAST_FMA** macro is defined if and only if the **fma** function is implemented directly with a hardware multiply-add instruction. Software implementations are expected to be substantially slower.

Returns

3 The **cbrt** functions return $x^{\frac{1}{3}}$.

~~Synopsis replace~~

```
#include <stdint.h>
#include <math.h>
double compoundn(double x, intmax_t n);
float compoundnf(float x, intmax_t n);
long double compoundnl(long double x, intmax_t n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 compoundnd32(_Decimal32 x, intmax_t n);
_Decimal64 compoundnd64(_Decimal64 x, intmax_t n);
_Decimal128 compoundnd128(_Decimal128 x, intmax_t n);
#endif
```

~~**Description**~~

~~The compute 1 plus x, raised to the power n. A domain error occurs if $x < -1$. A range error may occur if n is too large, depending on x. A pole error may occur if x equals -1 and $n < 0$.~~

~~**Returns**~~

~~The return $(1 + x)^n$.~~

7.12.7.2 The fabs functions

Synopsis

1

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fabsd32(_Decimal32 x);
_Decimal64 fabsd64(_Decimal64 x);
_Decimal128 fabsd128(_Decimal128 x);
#endif
```

Description

2 The **fabs** functions compute the absolute value of a floating-point number x.

Returns

3 The **fabs** functions return $|x|$.

7.12.7.3 The hypot functions

Synopsis

1

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 hypotd32(_Decimal32 x, _Decimal32 y);
_Decimal64 hypotd64(_Decimal64 x, _Decimal64 y);
_Decimal128 hypotd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **hypot** functions compute the square root of the sum of the squares of x and y, without undue overflow or underflow. A range error may occur.

3

Returns

4 The **hypot** functions return $\sqrt{x^2 + y^2}$.

7.12.7.4 The pow functions

Synopsis

1
```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
long double powl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 powd32(_Decimal32 x, _Decimal32 y);
_Decimal64 powd64(_Decimal64 x, _Decimal64 y);
_Decimal128 powd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

2 The **pow** functions compute x raised to the power y. A domain error occurs if x is finite and negative and y is finite and not an integer value. A range error may occur. A domain error may occur if x is zero and y is zero. A domain error or pole error may occur if x is zero and y is less than zero.

Returns

3 The **pow** functions return x^y.

~~Synopsis replace~~

```
#include <stdint.h>
#include <math.h>
double pown(double x, intmax_t n);
float pownf(float x, intmax_t n);
long double pownl(long double x, intmax_t n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 pownd32(_Decimal32 x, intmax_t n);
_Decimal64 pownd64(_Decimal64 x, intmax_t n);
_Decimal128 pownd128(_Decimal128 x, intmax_t n);
#endif
```

~~**Description**~~

~~The compute x raised to the th power. A range error may occur. A pole error may occur if x equals 0 and n < 0.~~

~~**Returns**~~

~~The return x^n.~~

7.12.7.5 The powr functions

Synopsis

1
```
#include <math.h>
double powr(double y, double x);
float powrf(float y, float x);
long double powrl(long double y, long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 powrd32(_Decimal32 y, _Decimal32 x);
_Decimal64 powrd64(_Decimal64 y, _Decimal64 x);
_Decimal128 powrd128(_Decimal128 y, _Decimal128 x);
#endif
```

Description

2 The **powr** functions compute x raised to the power y as $e^{y \log x}$. A domain error occurs if x < 0 or if x and y are both zero. A range error may occur. A pole error may occur if x equals zero and finite y < 0.

Returns

3 The **powr** functions return x^y.

~~Synopsis replace~~

```
#include <stdint.h>
#include <math.h>
double rootn(double x, intmax_t n);
float rootnf(float x, intmax_t n);
long double rootnl(long double x, intmax_t n);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 rootnd32(_Decimal32 x, intmax_t n);
_Decimal64 rootnd64(_Decimal64 x, intmax_t n);
_Decimal128 rootnd128(_Decimal128 x, intmax_t n);
#endif
```

~~**Description**~~

~~The compute the principal th root of x. A domain error occurs if n is 0 or if x < 0 and n is even. A range error may occur if n is -1. A pole error may occur if x equals zero and n < 0.~~

~~**Returns**~~

~~The return $x^{\frac{1}{n}}$.~~

7.12.7.6 The rsqrt functions

Synopsis

1

```
#include <math.h>
double rsqrt(double x);
float rsqrtf(float x);
long double rsqrtl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 rsqrtd32(_Decimal32 x);
_Decimal64 rsqrtd64(_Decimal64 x);
_Decimal128 rsqrtd128(_Decimal128 x);
#endif
```

Description

2 The **rsqrt** functions compute the reciprocal of the square root of the argument. A domain error occurs if the argument is less than zero. A pole error may occur if the argument equals zero.

Returns

3 The **rsqrt** functions return $\frac{1}{\sqrt{x}}$.

7.12.7.7 The sqrt functions

Synopsis

1

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 sqrtd32(_Decimal32 x);
_Decimal64 sqrtd64(_Decimal64 x);
_Decimal128 sqrtd128(_Decimal128 x);
```

```
long long int llroundd32(_Decimal32 x);
long long int llroundd64(_Decimal64 x);
long long int llroundd128(_Decimal128 x);
#endif
```

Description

2 The **lround** and **llround** functions round their argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the rounded value is outside the range of the return type, the numeric result is unspecified and a domain error or range error may occur.

Returns

3 The **lround** and **llround** functions return the rounded integer value.

7.12.9.8 The roundeven functions

Synopsis

1
```
#include <math.h>
double roundeven(double x);
float roundevenf(float x);
long double roundevenl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 roundevend32(_Decimal32 x);
_Decimal64 roundevend64(_Decimal64 x);
_Decimal128 roundevend128(_Decimal128 x);
#endif
```

Description

2 The **roundeven** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases to even (that is, to the nearest value that is an even integer), regardless of the current rounding direction.

Returns

3 The **roundeven** functions return the rounded integer value.

7.12.9.9 The trunc functions

Synopsis

1
```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double truncl(long double x);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 truncd32(_Decimal32 x);
_Decimal64 truncd64(_Decimal64 x);
_Decimal128 truncd128(_Decimal128 x);
#endif
```

Description

2 The **trunc** functions round their argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

Returns

3 The **trunc** functions return the truncated integer value.

~~Synopsis replace~~
~~**Description**~~

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

Returns

The fromfp and ufromfp functions return the rounded integer value.

Upward rounding of double x to type int, without raising the "inexact" floating-point exception, is achieved by

Synopsis replace
Description

The fromfpx and ufromfpx functions differ from the fromfp and ufromfp functions, respectively, only in that the fromfpx and ufromfpx functions raise the "inexact" floating-point exception if a rounded result not exceeding the specified width differs in value from the argument x.

Returns

The fromfpx and ufromfpx functions return the rounded integer value.

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

7.12.10 Remainder functions

7.12.10.1 The fmod functions

Synopsis

1
```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 fmodd32(_Decimal32 x, _Decimal32 y);
_Decimal64 fmodd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmodd128(_Decimal128 x, _Decimal128 y);
#endif
```

Description

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

Returns

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

7.12.10.2 The remainder functions

Synopsis

1
```
#include <math.h>
double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
#ifdef __STDC_IEC_60559_DFP__
```

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.[282] 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,[283] <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. 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, 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 If either of the types **int_least***N***_t** or **uint_least***N***_t** are provided, the other is provided, too, and they are the corresponding signed an unsigned types of each other. If the types **int_least***N***_t** and **int_least***M***_t** are provided for $N < M$, the width of the former is less than or equal to the width of the latter.

4 The following types are required:

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

[283] 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**

as are the types **int_least**N**_t** and **uint_least**N**_t** for all N for which the exact-width types **int**N**_t** and **uint**N**_t** are provided. 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[284)] 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 If either of the types **int_fast**N**_t** or **uint_fast**N**_t** are provided, the other is provided, too, and they are the corresponding signed an unsigned types of each other. If the types **int_fast**N**_t** and **int_fast**M**_t** are provided for $N < M$, the width of the former is less than or equal to the width of the latter.

4 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**

as are the types **int_fast**N**_t** and **uint_fast**N**_t** for all N for which the exact-width types **int**N**_t** and **uint**N**_t** are provided. 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: The following type designates~~ The types **intmax_t** and **uintmax_t** designate a signed and an unsigned integer type, respectively, that are at least as wide as any basic integer type, the types

char16_t	**int_least64_t**	**size_t**	**wchar_t**
char32_t	**ptrdiff_t**	**uint_fast64_t**	**wint_t**
int_fast64_t	**sig_atomic_t**	**uint_least64_t**	

[284)] 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.

and, provided they exist, **intptr_t** and **uintptr_t**. These types are required.

2 The types **int_leastmax_t** and **uint_leastmax_t** designate a signed and an unsigned integer type~~capable of representing any value of any unsigned integer type:~~ , respectively, that are at least as wide as any integer type defined by the header <stdint.h> . These types are required.

3 **NOTE 1** The **intmax_t** and **uintmax_t** types are intended to provide a fallback for applications that deal with integers for which they lack specific type information. This mainly occurs for two different situations. First, for integers that appear in conditional inclusion (**#if** expressions, 6.10.1) they provide fallback types that capture the implementation specific capabilities during translation phase 4. Second, for some semantic type definitions that resolve to implementation specific types there are no special provisions for **printf**, **scanf** or similar functions. In particular, the **intmax_t** and **uintmax_t** types are intended to represent values of all types listed above and also the exact-width integer types for all $N \leq 64$.

4 **NOTE 2** Extended integer types that are not referred by the above list and that are wider than **signed long long int** may also be wider than **intmax_t**. The types **intmax_t** and **int_leastmax_t** may then be different.

5 **NOTE 3** The **int_leastmax_t** and **uint_leastmax_t** types are intended to provide a fallback for applications that deal with unknown integer types that are potentially wider than **intmax_t** or **uintmax_t**.

6 **EXAMPLE** An implementation that has historically fixed its type **intmax_t** to a 64 bit type, and seeks to add a 128 bit integer exact-width type to its extended integer types, may do so by providing types **uint128_t**, **uint_least128_t**, **uint_least128_t**, **uint_leastmax_t** and the corresponding signed types and macros of <stdint.h> and <inttypes.h> (7.8.1) without breaking binary compatibility.

Application code can then query the type and print it by using the appropriate macros:

```
~~uintmax_t~~
#include <stdint.h>
#include <stdio.h>
#include <inttypes.h>
#ifdef UINT128_MAX
typedef uint128_t bitset;
#else
typedef uint_least64_t bitset;
#endif
int main(void) {
  bitset all = -1;
  printf("the largest set is %#" PRIXLEASTMAX "\n", (uint_leastmax_t)all);
}
```

~~These types are required.~~

Recommended practice

7 Unless some **typedef** in the library clause enforces otherwise, it is recommended to resolve **intmax_t** to **signed long int** or **signed long long int**. It is recommended that the same set of integer literals is consistently accepted by all compilation phases, even if **intmax_t** is chosen to be wider than **signed long long int**. Implementations and applications should not use the types **int_leastmax_t** and **uint_leastmax_t** to describe application programmable interfaces.[285)]

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~~ Unless specified otherwise, 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.[286)]

7.20.2.1 Width of exact-width integer types

1

[285)] This document does not use them further in any of its clauses.

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

INT*N*_WIDTH	exactly *N*
UINT*N*_WIDTH	exactly *N*

7.20.2.2 Width of minimum-width 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

1

INT_FAST*N*_WIDTH	exactly **UINT_FAST*N*_WIDTH**
UINT_FAST*N*_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
INT_LEASTMAX_WIDTH	exactly **UINT_LEASTMAX_WIDTH**
UINT_LEASTMAX_WIDTH	/* see above */

7.20.3 Width of other integer types

1 The following object-like macros specify the 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. 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.[287)]

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

7.20.3.4 Width of **wchar_t**

1

WCHAR_WIDTH	8

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

7.20.3.5 Width of `wint_t`

1
```
WINT_WIDTH                      16
```

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. For types wider than **uintmax_t**, the macros shall only be defined if the implementation provides integer literals for the type that are suitable to be used in **#if** preprocessing directives. Otherwise, the definition of these macros is mandatory for any of the types that are provided by the implementation.

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~~ If defined, the macro **INT***N***_C**(*value*) expands to an integer constant expression corresponding to the promoted type **int_least***N***_t**. ~~The~~ If defined, the macro **UINT***N***_C**(*value*) expands to an integer constant expression corresponding to the promoted type **uint_least***N***_t**. ~~For example, if~~

2 **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)
```

2 If defined, the following macro expands to an integer constant expression having the value specified by its argument and the type **int_leastmax_t**:

```
INT_LEASTMAX_C(value)
```

If defined, the following macro expands to an integer constant expression having the value specified by its argument and the type **uint_leastmax_t**:

```
UINT_LEASTMAX_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. For types wider than **uintmax_t** and for which the corresponding minimum width integer constant macro with suffix **_C** is not defined, 7.20.4.1 and 7.20.4.2, the macros are not necessarily suitable to be used in **#if** preprocessing directives.

2 **EXAMPLE** If **UINTMAX_WIDTH** is 64, no literals wider than 64 exist, and **int128_t** and **uint128_t** are names for extend integer types, then the following are valid definitions.

```
#define UINT128_MAX (~((uint128_t)+0u))
#define INT128_MAX (((((int128_t)+1)<<126)-1)*2+1)
#define INT128_MIN (-INT128_MAX-1)
```

Nevertheless, in **#if** expressions these macros expand to valid integer expressions but will only evaluate to the same values as **UINT64_MAX**, **INT64_MAX** and **INT64_MIN**, respectively.

Constraints

3 If *N* is greater than **UINTMAX_WIDTH** and the corresponding macros **INT***N***_C** or **UINT***N***_C** are not defined, the derived **_MIN** and **_MAX** macros for the exact-width, minimum-width and fastest minimum-width types for *N* shall not be used in an **#if** preprocessing directive, unless they are operand of the **defined** operator.

4 If the macros **INT_LEASTMAX_C** and **UINT_LEASTMAX_C** are not defined, the derived macros **INT_LEASTMAX_MIN**, **INT_LEASTMAX_MAX** and **UINT_LEASTMAX_MAX** shall not be used in an **#if** preprocessing directive, unless they are operand of the **defined** operator.[288)]

Recommended practice

5 Because of the above constraints, applications should prefer the **_WIDTH** macros over the **_MIN** or **_MAX** macros for feature tests in **#if** preprocessing directives.

[288)] This constraint reflects the fact that these macros may have numerical values that exceed the largest value that is representable during preprocessing. In that case these constants will generally be expressed by constant expressions that are more complex and not suitable for preprocessing.

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

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

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

acospi	erf	fma	llrint	lround	~~rootn~~roundeven
asinpi	exp10m1	fminmag	llround	nearbyint	round
atan2pi	exp10	fmin	log10p1	nextafter	rsqrt
atan2	exp2m1	fmod	log10	nextdown	scalbln
atanpi	exp2	frexp	log1p	nexttoward	scalbn
cbrt	expm1	~~fromfpxfromfp~~	~~llogb~~ log2p1	nextup	sinpi
ceil	fdim	ilogb	log2	~~pown~~powr	tanpi
~~compoundn~~copysign	~~fm~~or	ldexp	logb	remainder	tgamma
cospi	fmaxmag	lgamma	logp1	remquo	trunc
erfc	fmax	llogb	lrint	rint	~~ufromfpxufromfp~~

If all arguments for generic parameters are real, then use of the macro invokes a real function (provided <math.h> defines a function of the determined type); otherwise, use of the macro is undefined.

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

carg	cimag	conj	cproj	creal

Use of the macro with any argument of standard floating or complex type invokes a complex function. Use of the macro with an argument of decimal floating type is undefined.

11 The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any suffix from the function names. Thus, the macros with f or d prefix are:

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

and the macros with d32 or d64 prefix are:

15 **EXAMPLE** With the declarations

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
#ifdef __STDC_IEC_60559_DFP__
_Decimal32 d32;
_Decimal64 d64;
_Decimal128 d128;
#endif
```

functions invoked by use of type-generic macros are shown in the following table:

macro use	invokation
`exp(n)`	`exp(n)`, the function
`acosh(f)`	`acoshf(f)`
`sin(d)`	`sin(d)`, the function
`atan(ld)`	`atanl(ld)`
`log(fc)`	`clogf(fc)`
`sqrt(dc)`	`csqrt(dc)`
`pow(ldc, f)`	`cpowl(ldc, f)`
`remainder(n, n)`	`remainder(n, n)`, the function
`nextafter(d, f)`	`nextafter(d, f)`, the function
`nexttoward(f, ld)`	`nexttowardf(f, ld)`
`copysign(n, ld)`	`copysignl(n, ld)`
`ceil(fc)`	undefined
`rint(dc)`	undefined
`fmax(ldc, ld)`	undefined
`carg(n)`	`carg(n)`, the function
`cproj(f)`	`cprojf(f)`
`creal(d)`	`creal(d)`, the function
`cimag(ld)`	`cimagl(ld)`
`fabs(fc)`	`cabsf(fc)`
`carg(dc)`	`carg(dc)`, the function
`cproj(ldc)`	`cprojl(ldc)`
`fsub(f, ld)`	`fsubl(f, ld)`
`fdiv(d, n)`	`fdiv(d, n)`, the function
`dfma(f, d, ld)`	`dfmal(f, d, ld)`
`dadd(f, f)`	`daddl(f, f)`
`dsqrt(dc)`	undefined
`exp(d64)`	`expd64(d64)`
`sqrt(d32)`	`sqrtd32(d32)`
`fmax(d64, d128)`	`fmaxd128(d64, d128)`
`pow(d32, n)`	`powd64(d32, n)`
`remainder(d64, d)`	undefined
`creal(d64)`	undefined
`remquo(d32, d32, &n)`	undefined
`llquantexp(d)`	undefined
`quantize(dc)`	undefined
`samequantum(n, n)`	undefined
`d32sub(d32, d128)`	`d32subd128(d32, d128)`
`d32div(d64, n)`	`d32divd64(d64, n)`
`d64fma(d32, d64, d128)`	`d64fmad128(d32, d64, d128)`
`d64add(d32, d32)`	`d64addd128(d32, d32)`
`d64sqrt(d)`	undefined
`dadd(n, d64)`	undefined

7.25.1 Integer-power type-generic macros

1 In the following specifications the type T of the first prototype parameter x corresponds to a real floating point type that is supported by the implementation, or to an implementation-defined set

of complex floating types. IT corresponds to a signed integer type that comprises all admissible values for the operation on type T. A macro call with first argument X then has the same effects as if a function with the indicated prototype for TX would be called, where TX is the type of X if that has a floating point type, or **double** otherwise.

Constraints

2 The first argument to a call of these macros shall correspond to an expression that has either one of the supported types specified for T or has an integer type. The second argument shall be an integer expression; if it has a value that cannot be converted to the type IT, the behavior is undefined.

7.25.1.1 The compoundn type-generic macro

Synopsis

1
```
#include <tgmath.h>
T compoundn(T x, IT n);
```

Description

2 The **compoundn** type-generic macro computes 1 plus x, raised to the power n. A domain error occurs if $x < -1$. A range error may occur if n is too large, depending on x. A pole error may occur if x equals -1 and $n < 0$.

Returns

3 The **compoundn** type-generic macro returns $(1 + x)^n$.

7.25.1.2 The pown type-generic macro

Synopsis

1
```
#include <tgmath.h>
T pown(T x, IT n);
```

Description

2 The **pown** type-generic macro computes x raised to the n^{th} power. A range error may occur. A pole error may occur if x equals 0 and $n < 0$.

Returns

3 The **pown** type-generic macro returns x^n.

7.25.1.3 The rootn type-generic macro

Synopsis

1
```
#include <tgmath.h>
T rootn(T x, IT n);
```

Description

2 The **rootn** type-generic macro computes the principal n^{th} root of x. A domain error occurs if n is 0 or if $x < 0$ and n is even. A range error may occur if n is -1. A pole error may occur if x equals zero and $n < 0$.

Returns

3 The **rootn** type-generic macro returns $x^{\frac{1}{n}}$ with the same type as x if that type is a floating point type or **double** otherwise.

7.25.2 Nearest integer type-generic macros

1 In the following specifications the type T of the first prototype parameter x corresponds to a real floating point type that is supported by the implementation. The return types ST and UT correspond to signed and unsigned integer types, respectively, that are at least as wide as **long long**. A macro call with first argument X then has the same effects as if a function with the indicated prototype for

TX would be called, where TX is the type of X if that has a floating point type, or **double** otherwise.

Constraints

2 The first argument to a call of these macros shall correspond to an expression that has one of the supported types specified for T or has an integer type.

7.25.2.1 The toint and touint type-generic macros

Synopsis

1
```
#include <tgmath.h>
ST toint(T x, int round, unsigned int width);
UT touint(T x, int round, unsigned int width);
```

Description

2 The **toint** and **touint** type-generic macros round x, using the math rounding direction indicated by **round**, to a signed or unsigned integer, respectively, of width bits, and return the result value in the integer type designated by ST or UT, respectively. If the value of the round argument is not equal to the value of a math rounding direction macro, the direction of rounding is unspecified. If the value of width exceeds the width of the return type, the rounding is to the full width of the return type. The **toint** and **touint** type-generic macros do not raise the "inexact" floating-point exception. If x is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width, or if width is zero, the type-generic macros return an unspecified value and a domain error occurs.

Returns

3 The **toint** and **touint** type-generic macros return the rounded integer value.

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

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

7.25.2.2 The tointx and touintx type-generic macros

Synopsis

1
```
#include <tgmath.h>
ST tointx(T x, int round, unsigned int width);
UT touintx(T x, int round, unsigned int width);
```

Description

2 The **tointx** and **touintx** type-generic macros differ from the **toint** and **touint** type-generic macros, respectively, only in that the **tointx** and **touintx** type-generic macros raise the "inexact" floating-point exception if a rounded result not exceeding the specified width differs in value from the argument x.

Returns

3 The **tointx** and **touintx** type-generic macros return the rounded integer value.

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

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

7.31 Future library directions

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

7.31.1 Complex arithmetic <complex.h>

1 The function names

cacospi	cexp10m1	clog10	~~crootn~~
casinpi	cexp10	clog1p	crsqrt
catanpi	cexp2m1	clog2p1	csinpi
~~ccompoundn~~	cexp2	clog2	ctanpi
ccospi	cexpm1	clogp1	ctgamma
cerfc	clgamma	~~cpown~~	
cerf	clog10p1	cpowr	

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

7.31.2 Character handling <ctype.h>

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

7.31.3 Errors <errno.h>

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

7.31.4 Floating-point environment <fenv.h>

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

7.31.5 Characteristics of floating types <float.h>

1 Macros that begin with DBL_, DEC32_, DEC64_, DEC128_, DEC_, FLT_, or LDBL_ and an uppercase letter may be added to the macros defined in the <float.h> header.

7.31.6 Format conversion of integer types <inttypes.h>

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

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

3 The type imaxdiv_t and the functions imaxabs, imaxdiv, strtoimax, strtoumax, wcstoimax and wcstoumax are obsolescent features.

7.31.7 Localization <locale.h>

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

7.31.8 Mathematics <math.h>

1 Macros that begin with FP_ or MATH_ and an uppercase letter may be added to the macros defined in the <math.h> header.

2 Use of the DECIMAL_DIG macro is an obsolescent feature. A similar type-specific macro, such as LDBL_DECIMAL_DIG, can be used instead.

3 Function names that begin with is and a lowercase letter may be added to the declarations in the <math.h> header.

4 The function names

cacosh	cratanh	crexp10	crlog1p	~~crrootn~~
cacospi	cratanpi	crexp2m1	crlog2p1	crrsqrt
cacos	cratan	crexp2	crlog2	crsinh
casinh	~~crcompoundn~~	crexpm1	crlogp1	crsinpi
casinpi	crcosh	crexp	crlog	crsin
casin	crcospi	crhypot	~~crpown~~	crtanh
cratan2pi	crcos	crlog10p1	crpowr	crtanpi
cratan2	crexp10m1	crlog10	crpow	crtan

and the same names suffixed with **f**, **l**, **d32**, **d64**, or **d128** may be added to the <math.h> header. The **cr** prefix is intended to indicate a correctly rounded version of the function.

7.31.9 Signal handling <signal.h>

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

7.31.10 Atomics <stdatomic.h>

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

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

7.31.11 Boolean type and values <stdbool.h>

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

7.31.12 Integer types <stdint.h>

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

7.31.13 Input/output <stdio.h>

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

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

7.31.14 General utilities <stdlib.h>

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

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

7.31.15 String handling <string.h>

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

7.31.16 Date and time <time.h>

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

acos	sqrt	exp2m1	lgamma	nextdown	trunc
asin	fabs	exp2	llogb	nexttoward	ufromfpxufromfp
atan	acospi	expm1	llrint	nextup	fadd
acosh	asinpi	fdim	llround	pownpowr	dadd
asinh	atan2pi	floor	log10p1	remainder	fsub
atanh	atan2	fmaxmag	log10	remquo	dsub
cos	atanpi	fmax	log1p	rint	fmul
sin	cbrt	fma	log2p1	rootnroundeven	dmul
tan	ceil	fminmag	log2	round	fdiv
cosh	compoundncopysign	fmin	logb	rsqrt	ddiv
sinh	cospi	fmod	logp1	scalbln	ffma
tanh	erfc	frexp	lrint	scalbn	dfma
exp	erf	fromfpxfromfphypot	lround	sinpi	fsqrt
log	exp10m1	ilogb	nearbyint	tanpi	dsqrt
pow	exp10	ldexp	nextafter	tgamma	

Only if the implementation does not define __STDC_NO_COMPLEX__:

carg	cimag	conj	cproj	creal

Only if the implementation defines __STDC_IEC_60559_DFP__:

d32add	d64sub	d32div	d64fma	quantize	llquantexp
d64add	d32mul	d64div	d32sqrt	samequantum	
d32sub	d64mul	d32fma	d64sqrt	quantum	

B.25 Threads <threads.h>

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

```
void call_once(once_flag *flag, void (*func)(void));
int cnd_broadcast(cnd_t *cond);
void cnd_destroy(cnd_t *cond);
int cnd_init(cnd_t *cond);
int cnd_signal(cnd_t *cond);
int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
      const struct timespec *restrict ts);
int cnd_wait(cnd_t *cond, mtx_t *mtx);
void mtx_destroy(mtx_t *mtx);
int mtx_init(mtx_t *mtx, int type);
int mtx_lock(mtx_t *mtx);
int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
int mtx_trylock(mtx_t *mtx);
int mtx_unlock(mtx_t *mtx);
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
thrd_t thrd_current(void);
int thrd_detach(thrd_t thr);
int thrd_equal(thrd_t thr0, thrd_t thr1);
_Noreturn void thrd_exit(int res);
int thrd_join(thrd_t thr, int *res);
```

roundToIntegralExact	**rint**	7.12.9.4, F.10.6.4
nextUp	**nextup**	7.12.11.5, F.10.8.5
nextDown	**nextdown**	7.12.11.6, F.10.8.6
remainder	**remainder**, **remquo**	7.12.10.2, F.10.7.2, 7.12.10.3, F.10.7.3
minNum	**fmin**	7.12.12.3, F.10.9.3
maxNum	**fmax**	7.12.12.2, F.10.9.2
minNumMag	**fminmag**	7.12.12.5, F.10.9.5
maxNumMag	**fmaxmag**	7.12.12.4, F.10.9.4
scaleB	**scalbn**, **scalbln**	7.12.6.19, F.10.3.19
logB	**logb**, **ilogb**, **llogb**	7.12.6.17, F.10.3.17, 7.12.6.8, F.10.3.8, 7.12.6.10, F.10.3.10
addition	+, **fadd**, **faddl**, **daddl**	6.5.6, 7.12.14.1, F.10.11
subtraction	- , **fsub**, **fsubl**, **dsubl**	6.5.6, 7.12.14.2, F.10.11
multiplication	*, **fmul**, **fmull**, **dmull**	6.5.5, 7.12.14.3, F.10.11
division	/, **fdiv**, **fdivl**, **ddivl**	6.5.5, 7.12.14.4, F.10.11
squareRoot	**sqrt**, **fsqrt**, **fsqrtl**, **dsqrtl**	7.12.7.7, F.10.4.10, 7.12.14.6, F.10.11
fusedMultiplyAdd	**fma**, **ffma**, **ffmal**, **dfmal**	7.12.13.1, F.10.10.1, 7.12.14.5, F.10.11
convertFromInt	cast and implicit conversion	6.3.1.4, 6.5.4
convertToIntegerTiesToEven convertToIntegerTowardZero convertToIntegerTowardPositive convertToIntegerTowardNegative	~~**fromfp**~~, ~~**ufromfp**~~ **toint**, **touint**	7.25.2.1, F.10.6.10
convertToIntegerTiesToAway	~~**fromfp**~~, ~~**ufromfp**~~ **toint**, **touint** , **lround**, **llround**	7.25.2.1, F.10.6.10, 7.12.9.7, F.10.6.7
convertToIntegerExactTiesToEven convertToIntegerExactTowardZero convertToIntegerExactTowardPositive convertToIntegerExactTowardNegative convertToIntegerExactTiesToAway	~~**fromfpx**~~, ~~**ufromfpx**~~ **tointx**, **touintx**	7.25.2.2, F.10.6.11
convertFormat - different formats	cast and implicit conversions	6.3.1.5, 6.5.4
convertFormat - same format	**canonicalize**	7.12.11.7, F.10.8.7
convertFromDecimalCharacter	**strtod**, **wcstod**, **scanf**, **wscanf**, decimal floating constants	7.22.1.5, 7.29.4.1.1, 7.21.6.4, 7.29.2.12, F.5
convertToDecimalCharacter	**printf**, **wprintf**, **strfromd**	7.21.6.3, 7.29.2.11, 7.22.1.3, F.5
convertFromHexCharacter	**strtod**, **wcstod**, **scanf**, **wscanf**, hexadecimal floating constants	7.22.1.5, 7.29.4.1.1, 7.21.6.4, 7.29.2.12, F.5
convertToHexCharacter	**printf**, **wprintf**, **strfromd**	7.21.6.3, 7.29.2.11, 7.22.1.3, F.5
copy	**memcpy**, **memmove**	7.24.2.1, 7.24.2.3
negate	-(x)	6.5.3.3
abs	**fabs**	7.12.7.2, F.10.4.3
copySign	**copysign**	7.12.11.1, F.10.8.1
compareQuietEqual	==	6.5.9, F.9.3

17 The integer constant `10` provides the radix operation defined in IEC 60559 for decimal floating-point arithmetic.

18 The **`samequantumd`***N* functions (7.12.15.2) provide the sameQuantum operation defined in IEC 60559 for decimal floating-point arithmetic.

19 The **`fe_dec_getround`** (7.6.5.3) and **`fe_dec_setround`** (7.6.5.6) functions provide the getDecimalRoundingDirection and setDecimalRoundingDirection operations defined in IEC 60559 for decimal floating-point arithmetic. The macros (7.6) **`FE_DEC_DOWNWARD`**, **`FE_DEC_TONEAREST`**, **`FE_DEC_TONEARESTFROMZERO`**, **`FE_DEC_TOWARDZERO`**, and **`FE_DEC_UPWARD`**, which are used in conjunction with the **`fe_dec_getround`** and **`fe_dec_setround`** functions, represent the IEC 60559 rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTiesToAway, roundTowardZero, and roundTowardPositive, respectively.

20 The **`quantumd`***N* (7.12.15.3) and **`llquantexpd`***N* (7.12.15.4) functions compute the quantum and the (quantum) exponent q defined in IEC 60559 for decimal numbers viewed as having integer significands.

21 The **`encodedecd`***N* (7.12.16.1) and **`decodedecd`***N* (7.12.16.2) functions provide the encodeDecimal and decodeDecimal operations defined in IEC 60559 for decimal floating-point arithmetic.

22 The **`encodebind`***N* (7.12.16.3) and **`decodebind`***N* (7.12.16.4) functions provide the encodeBinary and decodeBinary operations defined in IEC 60559 for decimal floating-point arithmetic.

23 The C functions and type-generic macros in the following table provide operations recommended by IEC 60559 and similar operations. Correct rounding, which IEC 60559 specifies for its operations, is not required for the C functions in the table. See also 7.31.8.

IEC 60559 operation	C function or macro	Clause
exp	**`exp`**	7.12.6.1, F.10.3.1
expm1	**`expm1`**	7.12.6.6, F.10.3.6
exp2	**`exp2`**	7.12.6.4, F.10.3.4
exp2m1	**`exp2m1`**	7.12.6.5, F.10.3.5
exp10	**`exp10`**	7.12.6.2, F.10.3.2
exp10m1	**`exp10m1`**	7.12.6.3, F.10.3.3
log	**`log`**	7.12.6.11, F.10.3.11
log2	**`log2`**	7.12.6.15, F.10.3.15
log10	**`log10`**	7.12.6.12, F.10.3.12
logp1	**`log1p`**, **`logp1`**	7.12.6.14, F.10.3.14
log2p1	**`log2p1`**	7.12.6.16, F.10.3.16
log10p1	**`log10p1`**	7.12.6.13, F.10.3.13
hypot	**`hypot`**	7.12.7.3, F.10.4.4
rSqrt	**`rsqrt`**	7.12.7.6, F.10.4.9
compound	**`compoundn`**	7.25.1.1, F.10.4.2
rootn	**`rootn`**	7.25.1.3, F.10.4.8
pown	**`pown`**	7.25.1.2, F.10.4.6
pow	**`pow`**	7.12.7.4, F.10.4.5
powr	**`powr`**	7.12.7.5, F.10.4.7
sin	**`sin`**	7.12.4.6, F.10.1.6
cos	**`cos`**	7.12.4.5, F.10.1.5
tan	**`tan`**	7.12.4.7, F.10.1.7
sinPi	**`sinpi`**	7.12.4.13, F.10.1.13
cosPi	**`cospi`**	7.12.4.12, F.10.1.12
	`tanpi`	7.12.4.14, F.10.1.14
	`asinpi`	7.12.4.9, F.10.1.9
	`acospi`	7.12.4.8, F.10.1.8
atanPi	**`atanpi`**	7.12.4.10, F.10.1.10
atan2Pi	**`atan2pi`**	7.12.4.11, F.10.1.11
asin	**`asin`**	7.12.4.2, F.10.1.2
... continued ...		

... continued ...		
IEC 60559 operation	C function	Clause
acos	**acos**	7.12.4.1, F.10.1.1
atan	**atan**	7.12.4.3, F.10.1.3
atan2	**atan2**	7.12.4.4, F.10.1.4
sinh	**sinh**	7.12.5.5, F.10.2.5
cosh	**cosh**	7.12.5.4, F.10.2.4
tanh	**tanh**	7.12.5.6, F.10.2.6
asinh	**asinh**	7.12.5.2, F.10.2.2
acosh	**acosh**	7.12.5.1, F.10.2.1
atanh	**atanh**	7.12.5.3, F.10.2.3

F.4 Floating to integer conversion

1 If the integer type is **_Bool**, 6.3.1.2 applies and the conversion raises no floating-point exceptions if the floating-point value is not a signaling NaN. Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the "invalid" floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by 6.3.1.4. Conversion of an integral floating value that does not exceed the range of the integer type raises no floating-point exceptions; whether conversion of a non-integral floating value raises the "inexact" floating-point exception is unspecified.[389)]

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

1 The <float.h> header defines the macro

```
CR_DECIMAL_DIG
```

if and only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where <float.h> is first included. If defined, **CR_DECIMAL_DIG** expands to an integral constant expression suitable for use in **#if** preprocessing directives whose value is a number such that conversions between all supported IEC 60559 binary formats and character sequences with at most **CR_DECIMAL_DIG** significant decimal digits are correctly rounded. The value of **CR_DECIMAL_DIG** shall be at least $M + 3$, where M is the maximum value of the *T*_**DECIMAL_DIG** macros for IEC 60559 binary formats. If the implementation correctly rounds for all numbers of significant decimal digits, then **CR_DECIMAL_DIG** shall have the value of the macro **UINTMAX_MAX**.

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

3 Conversions from character sequences with more than **CR_DECIMAL_DIG** significant decimal digits to types with IEC 60559 binary formats shall correctly round to an intermediate character sequence with **CR_DECIMAL_DIG** significant decimal digits, according to the applicable rounding direction, and correctly round the intermediate result (having **CR_DECIMAL_DIG** significant decimal digits) to the destination type. The "inexact" floating-point exception is raised (once) if either conversion is inexact.[390)] (The second conversion may raise the "overflow" or "underflow" floating-point exception.)

4 The specification in this subclause assures conversion between IEC 60559 binary format and decimal character sequence follows all pertinent recommended practice. It also assures conversion from IEC 60559 format to decimal character sequence with at least *T*_**DECIMAL_DIG** digits and back, using to-nearest rounding, is the identity function, where T is the macro prefix for the format.

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

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

F.10.6.8 The `roundeven` functions

1

— **roundeven**(± 0) returns ± 0.

— **roundeven**$(\pm \infty)$ returns $\pm \infty$.

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

3 See the sample implementation for **ceil** in F.10.6.1.

F.10.6.9 The `trunc` functions

1 The **trunc** functions use IEC 60559 rounding toward zero (regardless of the current rounding direction).

— **trunc**(± 0) returns ± 0.

— **trunc**$(\pm \infty)$ returns $\pm \infty$.

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

F.10.6.10 The `toint` and `touint` type-generic macros

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

2 These ~~functions~~ type-generic macros do not raise the "inexact" floating-point exception.

F.10.6.11 The `tointx` and `touintx` type-generic macros

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

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

F.10.7 Remainder functions

F.10.7.1 The `fmod` functions

1

— **fmod**$(\pm 0, y)$ returns ± 0 for y not zero.

— **fmod**(x, y) returns a NaN and raises the "invalid" floating-point exception for x infinite or y zero (and neither is a NaN).

— **fmod**$(x, \pm \infty)$ returns x for x not infinite.

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

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

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

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 ~~636~~ 672 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
```

```
toupper
towctrans
towlower
towupper
tss_create
tss_delete
tss_dtor_t
tss_get
tss_set
tss_t
UINT128_C
UINT128_MAX
uint128_t
UINT128_WIDTH
UINT16_C
UINT16_MAX
uint16_t
UINT16_WIDTH
UINT32_C
UINT32_MAX
uint32_t
UINT32_WIDTH
UINT64_C
UINT64_MAX
uint64_t
UINT64_WIDTH
UINT8_C
UINT8_MAX
uint8_t
UINT8_WIDTH
uint_fast128_t
uint_fast16_t
uint_fast32_t
uint_fast64_t
uint_fast8_t
uint_least128_t
uint_least16_t
uint_least32_t
uint_least64_t
uint_least8_t
UINT_LEASTMAX_C
UINT_LEASTMAX_MAX
uint_leastmax_t
UINT_LEASTMAX_WIDTH
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
```

J.6.2 Particular identifiers or keywords

1 The following ~~1190~~ 1149 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
```

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

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

— 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