



HAL
open science

Remove conditional "WANT" macros from numbered clauses proposal for C2x

Jens Gustedt

► **To cite this version:**

Jens Gustedt. Remove conditional "WANT" macros from numbered clauses proposal for C2x. [Technical Report] N2359, ISO JTC1/SC22/WG14. 2019. hal-02089861

HAL Id: hal-02089861

<https://inria.hal.science/hal-02089861>

Submitted on 4 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

April 4, 2019

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

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

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

1. INTRODUCTION

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

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

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

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

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

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

2. REMOVING DEPENDENCY FROM `__STDC_WANT_IEC_60559_BFP_EXT__`

The only construct in the standard that would be similar to `__STDC_WANT_IEC_60559_BFP_EXT__` is `__STDC_WANT_LIB_EXT1__` as it used by Annex K. Since the features of Annex K are optional (testable by `__STDC_LIB_EXT1__`) such a macro makes complete sense there, because we don't want an implementation that has Annex K to pollute the name space of all its users.

For the integration of TS 18661-1 the situation is different. It has mainly (see below) integrated directly into the body of the standard, and there is no reason (or feature test macro) that indicates that the interfaces should be optional. In the contrary, most of them are useful additions that should make coding with floating point data more convenient and numerical algorithms more robust.

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

Therefore we simply propose

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

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

3. ADDING VERSION TEST MACROS

The addition of about 150 new interfaces for a new C version can be quite a burden for large code bases that wish to migrate to C2x. Conflicts will not occur often, but they are likely to occur *somewhere* and should be easy to track and to manage.

Therefore we should provide an easy-to-use tool that allows for user code to control the possible damage, but on the other hand will not impose much of a maintenance burden for implementations either.

Another difficulty that appears when the community moves to a new C standard is the fact that nowadays compilers and C libraries often come from different hands, and thus their synchronization concerning a new standard is not trivial. History has shown that this has been mayor hurdle for early acceptance of new C standards, and that dependency of one single “language” version macro `__STDC_VERSION__` is not enough to clarify the situation. Therefore we propose to use a set of new macros of the form `__STDC_XXXX_VERSION__`. For example `<math.h>` sets a new macro `__STDC_MATH_VERSION__` to a value greater than `202000L`, and users can then test this as follows.

```
#include <math.h>
#if __STDC_MATH_VERSION__ > 202000L
#   error "this_code_likes_to_daddl,_fix_before_going_further"
#endif
```

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

4. REVISITING SOME OF THE NAMING CHOICES

Many of the new interfaces would better have been introduced with a name prefix, much as other headers did when they were added to the C standard. It seems that this opportunity has been missed, though I think that we still could take a turn and use names such as `fp_canonicalize` instead of `canonicalize`, `fp_add` instead of `fadd`, etc.

Where these additions are particularly bad is where they introduce a new naming scheme (without admitting it) that is even contraproductive to a future encapsulation of these interfaces in a type generic function. These are the functions

<code>fromfpf</code>	<code>fromfpxl</code>	<code>strfromd</code>	<code>ufromfpf</code>	<code>ufromfpxl</code>
<code>fromfpl</code>	<code>fromfpx</code>	<code>strfromf</code>	<code>ufromfpl</code>	<code>ufromfpx</code>
<code>fromfpxf</code>	<code>fromfp</code>	<code>strfroml</code>	<code>ufromfpxf</code>	<code>ufromfp</code>

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

Therefore we propose to rename these interfaces to names starting with the reserved prefix `to`, namely

<code>tointf</code>	<code>tointxl</code>	<code>tostrd</code>	<code>touintf</code>	<code>touintxl</code>
<code>tointl</code>	<code>tointx</code>	<code>tostrf</code>	<code>touintl</code>	<code>touintx</code>
<code>tointxf</code>	<code>toint</code>	<code>tostrl</code>	<code>touintxf</code>	<code>touint</code>

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

5. RESERVE ACTIVE PREFIXES FOR FUTURE USE

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

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

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

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

5.2.4.2 Numerical limits

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

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

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

- 1 The following identifiers are defined only if `__STDC_WANT_IEC_60559_BFP_EXT__` is defined as a macro at the point in the source file where `<limits.h>` is first included:

CHAR_WIDTH	SHRT_WIDTH	UINT_WIDTH	LLONG_WIDTH
SCHAR_WIDTH	USHRT_WIDTH	LONG_WIDTH	ULLONG_WIDTH
UCHAR_WIDTH	INT_WIDTH	ULONG_WIDTH	

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

¹⁹⁾See “future language directions” (6.11.3).

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

CHAR_BIT	8
-----------------	---

- minimum value for an object of type **signed char**

SCHAR_MIN	-127 // $-(2^7 - 1)$
------------------	----------------------

- maximum value for an object of type **signed char**

SCHAR_MAX	+127 // $2^7 - 1$
------------------	-------------------

- width of type **signed char**

SCHAR_WIDTH	8
--------------------	---

- maximum value for an object of type **unsigned char**

UCHAR_MAX	255 // $2^8 - 1$
------------------	------------------

- width of type **unsigned char**

UCHAR_WIDTH	8
--------------------	---

- minimum value for an object of type **char**

CHAR_MIN	<i>see below</i>
-----------------	------------------

- maximum value for an object of type **char**

CHAR_MAX	<i>see below</i>
-----------------	------------------

- width of type **char**

CHAR_WIDTH	8
-------------------	---

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

MB_LEN_MAX	1
-------------------	---

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

6.11 Future language directions

6.11.1 Floating types

- 1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **long double**.

6.11.2 Linkages of identifiers

- 1 Declaring an identifier with internal linkage at file scope without the **static** storage-class specifier is an obsolescent feature.

6.11.3 External names

- 1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

- 1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

6.11.5 Storage-class specifiers

- 1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

6.11.6 Function declarators

- 1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

6.11.7 Function definitions

- 1 The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

6.11.8 Pragma directives

- 1 Pragmas whose first preprocessing token is **STDC** are reserved for future standardization.

6.11.9 Predefined macro names

- 1 Macro names beginning with **__STDC__** are reserved for future standardization.
- 2 Uses of the **__STDC_IEC_559__** and **__STDC_IEC_559_COMPLEX__** macros are obsolescent features.

Returns

- 3 If the argument is a character for which **islower** is true and there are one or more corresponding characters, as specified by the current locale, for which **isupper** is true, the **toupper** function returns one of the corresponding characters (always the same one for any given locale); otherwise, the argument is returned unchanged.

represents the default dynamic floating-point environment — the one installed at program startup — and has type “pointer to const-qualified `fenv_t`”. It can be used as an argument to `<fenv.h>` functions that manage the dynamic floating-point environment.

- 14 Additional implementation-defined environments, with macro definitions beginning with `FE_` and an uppercase letter,²²³⁾ and having type “pointer to const-qualified `fenv_t`”, may also be specified by the implementation.

7.6.1 The `FENV_ACCESS` pragma

Synopsis

```
1  #include <fenv.h>
   #pragma STDC FENV_ACCESS on-off-switch
```

Description

- 2 The `FENV_ACCESS` pragma provides a means to inform the implementation when a program might access the floating-point environment to test floating-point status flags or run under non-default floating-point control modes.²²⁴⁾ The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FENV_ACCESS` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined. If part of a program tests floating-point status flags or establishes non-default floating-point mode settings using any means other than the `FENV_ROUND` pragmas, but was translated with the state for the `FENV_ACCESS` pragma “off”, the behavior is undefined. The default state (“on” or “off”) for the pragma is implementation-defined. (When execution passes from a part of the program translated with `FENV_ACCESS` “off” to a part translated with `FENV_ACCESS` “on”, the state of the floating-point status flags is unspecified and the floating-point control modes have their default settings.)

3 EXAMPLE

```
#include <fenv.h>
void f(double x)
{
    #pragma STDC FENV_ACCESS ON
    void g(double);
    void h(double);
    /* ... */
    g(x + 1);
    h(x + 1);
    /* ... */
}
```

- 4 If the function `g` might depend on status flags set as a side effect of the first `x + 1`, or if the second `x + 1` might depend on control modes set as a side effect of the call to function `g`, then the program has to contain an appropriately placed invocation of `#pragma STDC FENV_ACCESS ON` as shown.²²⁵⁾

7.6.2 The `FENV_ROUND` pragma

Synopsis

```
1  #define __STDC_WANT_IEC_60559_BFP_EXT__
```

²²³⁾See “future library directions” (7.31.4).

²²⁴⁾The purpose of the `FENV_ACCESS` pragma is to allow certain optimizations that could subvert flag tests and mode changes (e.g., global common subexpression elimination, code motion, and constant folding). In general, if the state of `FENV_ACCESS` is “off”, the translator can assume that the flags are not tested, and that default modes are in effect, except where specified otherwise by an `FENV_ROUND` pragma.

²²⁵⁾The side effects impose a temporal ordering that requires two evaluations of `x + 1`. On the other hand, without the `#pragma STDC FENV_ACCESS ON` pragma, and assuming the default state is “off”, just one evaluation of `x + 1` would suffice.

- 11 **EXAMPLE 2** The following table illustrates how the `cs_precedes`, `sep_by_space`, and `sign_posn` members affect the formatted value.

<code>p_cs_precedes</code>	<code>p_sign_posn</code>	<code>p_sep_by_space</code>		
		0	1	2
0	0	(1.25\$)	(1.25 \$)	(1.25\$)
	1	+1.25\$	+1.25 \$	+ 1.25\$
	2	1.25\$+	1.25 \$+	1.25\$ +
	3	1.25+\$	1.25 +\$	1.25+ \$
	4	1.25\$+	1.25 \$+	1.25\$ +
1	0	(\$1.25)	(\$ 1.25)	(\$1.25)
	1	+\$1.25	+\$ 1.25	+ \$1.25
	2	\$1.25+	\$ 1.25+	\$1.25 +
	3	+\$1.25	+\$ 1.25	+ \$1.25
	4	\$+1.25	\$+ 1.25	\$ +1.25

7.12 Mathematics <math.h>

- The header <math.h> declares two types and many mathematical functions and defines several macros. Most synopses specify a family of functions consisting of a principal function with one or more **double** parameters, a **double** return value, or both; and other functions with the same name but with **f** and **l** suffixes, which are corresponding functions with **float** and **long double** parameters, return values, or both.²³⁸⁾ Integer arithmetic functions and conversion functions are discussed later.
- The following identifiers are defined or declared only if `__STDC_WANT_IEC_60559_BFP_EXT__` is defined as a macro at the point in the source file where <math.h> is first included:

FP_INT_UPWARD	FP_FAST_FADD	FP_FAST_FDIVL
FP_INT_DOWNWARD	FP_FAST_FADDL	FP_FAST_DDIVL
FP_INT_TOWARDZERO	FP_FAST_DADDL	FP_FAST_FFMA
FP_INT_TONEARESTFROMZERO	FP_FAST_FSUB	FP_FAST_FFMAI
FP_INT_TONEAREST	FP_FAST_FSUBL	FP_FAST_DFMAL
FP_LLOGB0	FP_FAST_DSUBL	FP_FAST_FSQRT
FP_LLOGBNAN	FP_FAST_FMUL	FP_FAST_FSQRTL
SNANF	FP_FAST_FMULL	FP_FAST_DSQRTL
SNAN	FP_FAST_DMULL	
SNANL	FP_FAST_FDIV	

iseqsig	ufromfpxf	nextupf	fdivl
iscanonical	ufromfpxl	nextupl	ddivl
issignaling	roundeven	nextdown	ffma
issubnormal	roundevenf	nextdownf	ffmai
iszero	roundevenl	nextdownl	dfmai
fromfp	llogb	fadd	fsqrt
fromfpf	llogbf	faddl	fsqrtl
fromfpl	llogbl	daddl	dsqrtl
ufromfp	fmaxmag	fsub	canonicalize
ufromfpf	fmaxmagf	fsubl	canonicalizef
ufromfpl	fmaxmagl	dsubl	canonicalizel
fromfpx	fminmag	fmul	
fromfpxf	fminmagf	fmull	
fromfpxl	fminmagl	dmull	
ufromfpx	nextup	fdiv	

- The types

```
float_t
double_t
```

are floating types at least as wide as **float** and **double**, respectively, and such that **double_t** is at least as wide as **float_t**. If **FLT_EVAL_METHOD** equals 0, **float_t** and **double_t** are **float** and **double**, respectively; if **FLT_EVAL_METHOD** equals 1, they are both **double**; if **FLT_EVAL_METHOD** equals 2, they are both **long double**; and for other values of **FLT_EVAL_METHOD**, they are otherwise implementation-defined.²³⁹⁾

- The macro

```
HUGE_VAL
```

²³⁸⁾ Particularly on systems with wide expression evaluation, a <math.h> function might pass arguments and return values in wider format than the synopsis prototype indicates.

²³⁹⁾ The types **float_t** and **double_t** are intended to be the implementation's most efficient types at least as wide as **float** and **double**, respectively. For **FLT_EVAL_METHOD** equal 0, 1, or 2, the type **float_t** is the narrowest type used by the implementation to evaluate floating expressions.

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

```
HUGE_VALF
HUGE_VALL
```

are respectively **float** and **long double** analogs of **HUGE_VAL**.²⁴⁰⁾

- 5 The macro

```
INFINITY
```

expands to a constant expression of type **float** representing positive or unsigned infinity, if available; else to a positive constant of type **float** that overflows at translation time.²⁴¹⁾

- 6 The macro

```
NAN
```

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

- 7 The *signaling NaN macros*

```
SNANF
SNAN
SNANL
```

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

- 8 The *number classification macros*

```
FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO
```

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

- 9 The *math rounding direction macros*

```
FP_INT_UPWARD
FP_INT_DOWNWARD
FP_INT_TOWARDZERO
FP_INT_TONEARESTFROMZERO
FP_INT_TONEAREST
```

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

²⁴⁰⁾ **HUGE_VAL**, **HUGE_VALF**, and **HUGE_VALL** can be positive infinities in an implementation that supports infinities.

²⁴¹⁾ In this case, using **INFINITY** will violate the constraint in 6.4.4 and thus require a diagnostic.

Description

- 2 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 type²⁵²) of the specified width, or if **width** is zero, the functions return an unspecified value and a domain error occurs.

Returns

- 3 The **fromfp** and **ufromfp** functions return the rounded integer value.
- 4 **EXAMPLE** Upward rounding of **double** x to type **int**, without raising the “inexact” floating-point exception, is achieved by

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

7.12.9.11 The **fromfpx** and **ufromfpx** functions

Synopsis

- ```
1 #define __STDC_WANT_IEC_60559_BFP_EXT__
 #include <stdint.h>
 #include <math.h>
 intmax_t fromfpx(double x, int round, unsigned int width);
 intmax_t fromfpxf(float x, int round, unsigned int width);
 intmax_t fromfpxl(long double x, int round, unsigned int width);
 uintmax_t ufromfpx(double x, int round, unsigned int width);
 uintmax_t ufromfpxf(float x, int round, unsigned int width);
 uintmax_t ufromfpxl(long double x, int round, unsigned int width);
```

### Description

- 2 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

- 3 The **fromfpx** and **ufromfpx** functions return the rounded integer value.
- 4 **NOTE** Conversions to integer types that are not required to raise the inexact exception can be done simply by rounding to integral value in floating type and then converting to the target integer type. For example, the conversion of **long double**  $x$  to **uint64\_t**, using upward rounding, is done by

```
(uint64_t)ceil(x)
```

## 7.12.10 Remainder functions

### 7.12.10.1 The **fmod** functions

#### Synopsis

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

Description

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

²⁵²)For signed types, 6.2.6.2 permits three representations, which differ in whether a value of $-(2^M)$, where M is the number of value bits, can be represented.

Returns

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

7.12.10.2 The remainder functions**Synopsis**

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

Description

- 2 The **remainder** functions compute the remainder $x \text{ REM } y$ required by IEC 60559.²⁵³⁾

Returns

- 3 The **remainder** functions return $x \text{ REM } y$. If y is zero, whether a domain error occurs or the functions return zero is implementation defined.

7.12.10.3 The remquo functions**Synopsis**

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

Description

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

Returns

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

7.12.11 Manipulation functions**7.12.11.1 The copysign functions****Synopsis**

```
1  #include <math.h>
    double copysign(double x, double y);
    float copysignf(float x, float y);
    long double copysignl(long double x, long double y);
```

Description

- 2 The **copysign** functions produce a value with the magnitude of x and the sign of y . They produce a NaN (with the sign of y) if x is a NaN. On implementations that represent a signed zero but do not treat negative zero consistently in arithmetic operations, the **copysign** functions regard the sign of zero as positive.

²⁵³⁾“When $y \neq 0$, the remainder $r = x \text{ REM } y$ is defined regardless of the rounding mode by the mathematical relation $r = x - ny$, where n is the integer nearest the exact value of x/y ; whenever $|n - x/y| = 1/2$, then n is even. If $r = 0$, its sign shall be that of x .” This definition is applicable for all implementations.

7.20 Integer types <stdint.h>

- 1 The header <stdint.h> declares sets of integer types having specified widths, and defines corresponding sets of macros.²⁷⁹⁾ It also defines macros that specify limits of integer types corresponding to types defined in other standard headers.
- 2 Types are defined in the following categories:
 - integer types having certain exact widths;
 - integer types having at least certain specified widths;
 - fastest integer types having at least certain specified widths;
 - integer types wide enough to hold pointers to objects;
 - integer types having greatest width.

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

- 3 Corresponding macros specify limits of the declared types and construct suitable constants.
- 4 For each type described herein that the implementation provides,²⁸⁰⁾ <stdint.h> shall declare that typedef name and define the associated macros. Conversely, for each type described herein that the implementation does not provide, <stdint.h> shall not declare that typedef name nor shall it define the associated macros. An implementation shall provide those types described as “required”, but need not provide any of the others (described as “optional”).
- 5 The following identifiers are defined only if `__STDC_WANT_IEC_60559_BFP_EXT__` is defined as a macro at the point in the source file where <stdint.h> is first included:

INTN_WIDTH	INTPTR_WIDTH	SIZE_WIDTH
UINTN_WIDTH	UINTPTR_WIDTH	WCHAR_WIDTH
INT_LEASTN_WIDTH	INTMAX_WIDTH	WINT_WIDTH
UINT_LEASTN_WIDTH	UINTMAX_WIDTH	
INT_FASTN_WIDTH	PTRDIFF_WIDTH	
UINT_FASTN_WIDTH	SIG_ATOMIC_WIDTH	

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 `intN_t` designates a signed integer type with width *N*, no padding bits, and a two’s complement representation. Thus, `int8_t` denotes such a signed integer type with a width of exactly 8 bits.
- 2 The typedef name `uintN_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, no padding bits, and (for the signed types) that have a two’s complement representation, it shall define the corresponding typedef names.

²⁷⁹⁾See “future library directions” (7.31.12).

²⁸⁰⁾Some of these types might denote implementation-defined extended integer types.

7.22 General utilities <stdlib.h>

- 1 The header <stdlib.h> declares five types and several functions of general utility, and defines several macros.³¹¹⁾
- 2 The following identifiers are declared only if `__STDC_WANT_IEC_60559_BFP_EXT__` is defined as a macro at the point in the source file where <stdlib.h> is first included:

strfromd

strfromf

strfroml

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

```
div_t
```

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

```
ldiv_t
```

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

```
lldiv_t
```

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

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

```
EXIT_FAILURE
```

and

```
EXIT_SUCCESS
```

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

```
RAND_MAX
```

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

```
MB_CUR_MAX
```

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

7.22.1 Numeric conversion functions

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

7.22.1.1 The `atof` function

Synopsis

- 1


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

³¹¹⁾See “future library directions” (7.31.14).

Description

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

```
—— atof: strtod(nptr, (char **)NULL)
—— atof: strtod(nptr, nullptr)
```

Returns

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

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

7.22.1.2 The atoi, atol, and atoll functions**Synopsis**

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

**Description**

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

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

**Returns**

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

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

**7.22.1.3 The strfromd, strfromf, and strfroml functions****Synopsis**

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

Description

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

Returns

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

7.25 Type-generic math <tgmath.h>

- 1 The header <tgmath.h> includes the headers <math.h> and <complex.h> and defines several type-generic macros.
- 2 The following identifiers are defined as type-generic macros only if `__STDC_WANT_IEC_60559_BFP_EXT__` is defined as a macro at the point in the source file where <tgmath.h> is first included:

roundeven	nextup	fromfpx	fsub	fdiv	fsqrt
llogb	nextdown	ufromfpx	dsub	ddiv	dsqrt
fmaxmag	fromfp	fadd	fmul	ffma	
fminmag	ufromfp	dadd	dmul	dfma	

- 3 Of the <math.h> and <complex.h> functions without an **f** (**float**) or **l** (**long double**) suffix, several have one or more parameters whose corresponding real type is **double**. For each such function, except the functions that round result to narrower type (7.12.14) (which are covered below) and **modf**, there is a corresponding *type-generic macro*.³³²⁾ The parameters whose corresponding real type is **double** in the function synopsis are *generic parameters*. Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.³³³⁾
- 4 Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:
 - First, if any argument for generic parameters has type **long double**, the type determined is **long double**.
 - Otherwise, if any argument for generic parameters has type **double** or is of integer type, the type determined is **double**.
 - Otherwise, the type determined is **float**.
- 5 For each unsuffixed function in <math.h> for which there is a function in <complex.h> with the same name except for a **c** prefix, the corresponding type-generic macro (for both functions) has the same name as the function in <math.h>. The corresponding type-generic macro for **fabs** and **cabs** is **fabs**.

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

³³²⁾Like other function-like macros in standard libraries, each type-generic macro can be suppressed to make available the corresponding ordinary function.

³³³⁾If the type of the argument is not compatible with the type of the parameter for the selected function, the behavior is undefined.

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.

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

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

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

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

carg	cimag	conj	cproj	creal
-------------	--------------	-------------	--------------	--------------

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

- 8 The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any **l** suffix³³⁴⁾ from the function names. Thus, the macros are:

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

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

- 9 A type-generic macro corresponding to a function indicated in the table in 7.6.2 is affected by constant rounding modes (7.6.3).
- 10 **NOTE** The type-generic macro definition in the example in 6.5.1.1 does not conform to this specification. A conforming macro could be implemented as follows:

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

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

³³⁴⁾There are no functions with these macro names and the **f** suffix.

7.31 Future library directions

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

7.31.1 Complex arithmetic <complex.h>

- 1 The function names

cerf	cexpm1	clog2
cerfc	clog10	clgamma
cexp2	clog1p	ctgamma

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

7.31.2 Character handling <ctype.h>

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

7.31.3 Errors <errno.h>

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

7.31.4 Floating-point environment <fenv.h>

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

7.31.5 Format conversion of integer types <inttypes.h>

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

7.31.6 Localization <locale.h>

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

7.31.7 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.

7.31.8 Alignment <stdalign.h>

- 1 [The header <stdalign.h> together with its defined macros `__alignas_is_defined` and `__alignas_is_defined` is an obsolescent feature.](#)

7.31.9 Atomics <stdatomic.h>

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

7.31.10 Common definitions <stddef.h>

- 1 [The macro `NULL` 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~~ header <stdbool.h> ~~together with its defined macro __bool_true_false_are_defined~~ 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** 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.

7.31.17 Threads <threads.h>

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

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

- 1 Function names that begin with **wcs** and a lowercase letter may be added to the declarations in the <wchar.h> header.
- 2 Lowercase letters may be added to the conversion specifiers and length modifiers in **fwprintf** and **fwscanf**. Other characters may be used in extensions.

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

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

```

int abs(int j);
long int labs(long int j);
long long int llabs(long long int j);
div_t div(int numer, int denom);
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);
size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs, size_t n);

```

```

__STDC_WANT_LIB_EXT1__
errno_t
rsize_t
constraint_handler_t

constraint_handler_t set_constraint_handler_s(constraint_handler_t handler);
void abort_handler_s(const char *restrict msg, void *restrict ptr, errno_t error);
void ignore_handler_s(const char *restrict msg, void *restrict ptr, errno_t error);
errno_t getenv_s(
    size_t *restrict len, char *restrict value, rsize_t maxsize,
    const char *restrict name);
void *bsearch_s(
    const void *key, const void *base, rsize_t nmemb, rsize_t size,
    int (*compar)(const void *k, const void *y, void *context),
    void *context);
errno_t qsort_s(
    void *base, rsize_t nmemb, rsize_t size,
    int (*compar)(const void *x, const void *y, void *context),
    void *context);
errno_t wctomb_s(int *restrict status, char *restrict s, rsize_t smax, wchar_t wc);
errno_t mbstowcs_s(
    size_t *restrict retval, wchar_t *restrict dst, rsize_t dstmax,
    const char *restrict src, rsize_t len);
errno_t wcstombs_s(
    size_t *restrict retval, char *restrict dst, rsize_t dstmax,
    const wchar_t *restrict src, rsize_t len);

```

B.22 noreturn <stdnoreturn.h>

~~noreturn~~

[This header is empty.](#)

B.23 String handling <string.h>

size_t NULL

```

void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *restrict s1, const char *restrict s2);
char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
char *strcat(char *restrict s1, const char *restrict s2);
char *strncat(char *restrict s1, const char *restrict s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);
void *memchr(const void *s, int c, size_t n);

```

```
#define LDBL_DIG 10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP -37
#define LDBL_MIN_EXP
```

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

```
#define DBL_MAX 1E+37
#define FLT_MAX 1E+37
#define LDBL_MAX 1E+37
```

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

```
#define DBL_EPSILON 1E-9
#define DBL_MIN 1E-37
#define FLT_EPSILON 1E-5
#define FLT_MIN 1E-37
#define LDBL_EPSILON 1E-9
#define LDBL_MIN 1E-37
```

Annex F

(normative)

IEC 60559 floating-point arithmetic

F.1 Introduction

- 1 This annex specifies C language support for the IEC 60559 floating-point standard. The IEC 60559 floating-point standard is specifically *Floating-point arithmetic* (ISO/IEC/IEEE 60559:2011), also designated as *IEEE Standard for Floating-Point Arithmetic* (IEEE 754–2008). The IEC 60559 floating-point standard supersedes the IEC 60559:1989 *binary arithmetic standard*, also designated as *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754–1985). IEC 60559 generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc.
- 2 The IEC 60559 floating-point standard specifies decimal, as well as binary, floating-point arithmetic. It supersedes *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854–1987) which generalized the *binary arithmetic standard* (IEEE 754–1985) to remove dependencies on radix and word length.
- 3 An implementation that defines `__STDC_IEC_60559_BFP__` to `yyymmL` shall conform to the specifications in this annex and shall also define `__STDC_IEC_559__` to 1.³⁷⁷⁾ Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

F.2 Types

- 1 The C floating types match the IEC 60559 formats as follows:
 - The **float** type matches the IEC 60559 binary32 format.
 - The **double** type matches the IEC 60559 binary64 format.
 - The **long double** type matches the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format,³⁷⁸⁾ else a non-IEC 60559 extended format, else the IEC 60559 binary64 format.

Any non-IEC 60559 extended format used for the **long double** type shall have more precision than IEC 60559 binary64 and at least the range of IEC 60559 binary64.³⁷⁹⁾ The value of **FLT_ROUNDS** applies to all IEC 60559 types supported by the implementation, but need not apply to non-IEC 60559 types.

Recommended practice

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

F.2.1 Infinities and NaNs

- 1 Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values (5.2.4.2.2).
- 2 The **NAN** and **INFINITY** macros and the `nan` functions in `<math.h>` provide designations for IEC 60559 quiet NaNs and infinities. The **SNANF**, **SNAN**, and **SNANL** macros in `<math.h>` provide designations for IEC 60559 signaling NaNs.
- 3 This annex does not require the full support for signaling NaNs specified in IEC 60559. This annex uses the term NaN, unless explicitly qualified, to denote quiet NaNs. Where specification of signaling NaNs is not provided, the behavior of signaling NaNs is implementation-defined (either treated as an IEC 60559 quiet NaN or treated as an IEC 60559 signaling NaN).³⁸⁰⁾

³⁷⁷⁾Implementations that do not define either of `__STDC_IEC_60559_BFP__` and `__STDC_IEC_559__` are not required to conform to these specifications. New code should not use the obsolescent macro `__STDC_IEC_559__` to test for conformance to this annex.

³⁷⁸⁾IEC 60559 binary64-extended formats include the common 80-bit IEC 60559 format.

³⁷⁹⁾A non-IEC 60559 **long double** type is required to provide infinity and NaNs, as its values include all **double** values.

³⁸⁰⁾Since NaNs created by IEC 60559 operations are always quiet, quiet NaNs (along with infinities) are sufficient for closure of the arithmetic.

- 4 Any operator or `<math.h>` function that raises an “invalid” floating-point exception, if delivering a floating type result, shall return a quiet NaN.
- 5 In order to support signaling NaNs as specified in IEC 60559, an implementation should adhere to the following recommended practice.

Recommended practice

- 6 Any floating-point operator or `<math.h>` function or macro with a signaling NaN input, unless explicitly specified otherwise, raises an “invalid” floating-point exception.
- 7 **NOTE** Some functions do not propagate quiet NaN arguments. For example, `hypot(x, y)` returns infinity if `x` or `y` is infinite and the other is a quiet NaN. The recommended practice in this subclause specifies that such functions (and others) raise the “invalid” floating-point exception if an argument is a signaling NaN, which also implies they return a quiet NaN in these cases.
- 8 The `<fenv.h>` header defines the macro `FE_SNANS_ALWAYS_SIGNAL` if and only if the implementation follows the recommended practice in this subclause. If defined, `FE_SNANS_ALWAYS_SIGNAL` expands to the integer constant 1.

F.3 Operations

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

Operation binding

IEC 60559 operation	C operation	Clause
<code>roundToIntegralTiesToEven</code>	<code>roundeven</code>	7.12.9.8, F.10.6.8
<code>roundToIntegralTiesAway</code>	<code>round</code>	7.12.9.6, F.10.6.6
<code>roundToIntegralTowardZero</code>	<code>trunc</code>	7.12.9.9, F.10.6.9
<code>roundToIntegralTowardPositive</code>	<code>ceil</code>	7.12.9.1, F.10.6.1
<code>roundToIntegralTowardNegative</code>	<code>floor</code>	7.12.9.2, F.10.6.2
<code>roundToIntegralExact</code>	<code>rint</code>	7.12.9.4, F.10.6.4
<code>nextUp</code>	<code>nextup</code>	7.12.11.5, F.10.8.5
<code>nextDown</code>	<code>nextdown</code>	7.12.11.6, F.10.8.6
<code>remainder</code>	<code>remainder, remquo</code>	7.12.10.2, F.10.7.2, 7.12.10.3, F.10.7.3
<code>minNum</code>	<code>fmin</code>	7.12.12.3, F.10.9.3
<code>maxNum</code>	<code>fmax</code>	7.12.12.2, F.10.9.2
<code>minNumMag</code>	<code>fminmag</code>	7.12.12.5, F.10.9.5
<code>maxNumMag</code>	<code>fmaxmag</code>	7.12.12.4, F.10.9.4
<code>scaleB</code>	<code>scalbn, scalbln</code>	7.12.6.14, F.10.3.14
<code>logB</code>	<code>logb, ilogb, llogb</code>	7.12.6.12, F.10.3.12, 7.12.6.5, F.10.3.5, 7.12.6.7, F.10.3.7
<code>addition</code>	<code>+, fadd, faddl, daddl</code>	6.5.6, 7.12.14.1, F.10.11
<code>subtraction</code>	<code>-, fsub, fsubl, dsubl</code>	6.5.6, 7.12.14.2, F.10.11
<code>multiplication</code>	<code>*, fmul, fmul, dmul</code>	6.5.5, 7.12.14.3, F.10.11
<code>division</code>	<code>/, fdiv, fdivl, ddivl</code>	6.5.5, 7.12.14.4, F.10.11
<code>squareRoot</code>	<code>sqrt, fsqrt, fsqrtl, dsqrtl</code>	7.12.7.5, F.10.4.5, 7.12.14.6, F.10.11
<code>fusedMultiplyAdd</code>	<code>fma, ffma, fmal, dfmal</code>	7.12.13.1, F.10.10.1, 7.12.14.5, F.10.11
<code>convertFromInt</code>	cast and implicit conversion	6.3.1.4, 6.5.4

convertToIntegerTiesToEven convertToIntegerTowardZero convertToIntegerTowardPositive convertToIntegerTowardNegative	fromfp, ufromfp	7.12.9.10, F.10.6.10
convertToIntegerTiesToAway	fromfp, ufromfp, lround, llround	7.12.9.10, F.10.6.10, 7.12.9.7, F.10.6.7
convertToIntegerExactTiesToEven convertToIntegerExactTowardZero convertToIntegerExactTowardPositive convertToIntegerExactTowardNegative convertToIntegerExactTiesToAway	fromfpx, ufromfpx	7.12.9.11, 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	7.22.1.4, 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	7.22.1.4, 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, 7.24.2.2
negate	-(x)	6.5.3.3
abs	fabs	7.12.7.2, F.10.4.2
copySign	copysign	7.12.11.1, F.10.8.1
compareQuietEqual	==	6.5.9, F.9.3
compareQuietNotEqual	!=	6.5.9, F.9.3
compareSignalingEqual	iseqsig	7.12.15.7, F.10.14.1
compareSignalingGreater	>	6.5.8, F.9.3
compareSignalingGreaterEqual	>=	6.5.8, F.9.3
compareSignalingLess	<	6.5.8, F.9.3
compareSignalingLessEqual	<=	6.5.8, F.9.3
compareSignalingNotEqual	! iseqsig(x)	7.12.15.7, F.10.14.1
compareSignalingNotGreater	! (x > y)	6.5.8, F.9.3
compareSignalingLessUnordered	! (x >= y)	6.5.8, F.9.3
compareSignalingNotLess	! (x < y)	6.5.8, F.9.3
compareSignalingGreaterUnordered	! (x <= y)	6.5.8, F.9.3
compareQuietGreater	isgreater	7.12.15.1
compareQuietGreaterEqual	isgreaterequal	7.12.15.2
compareQuietLess	isless	7.12.15.3
compareQuietLessEqual	islessequal	7.12.15.4
compareQuietUnordered	isunordered	7.12.15.6
compareQuietNotGreater	! isgreater(x, y)	7.12.15.1
compareQuietLessUnordered	! isgreaterequal(x, y)	7.12.15.2
compareQuietNotLess	! isless(x, y)	7.12.15.3
compareQuietGreaterUnordered	! islessequal(x, y)	7.12.15.4
compareQuietOrdered	! isunordered(x, y)	7.12.15.6
class	fpclassify, signbit, isignaling	7.12.3.1, 7.12.3.7, 7.12.3.8
isSignMinus	signbit	7.12.3.7
isNormal	isnormal	7.12.3.6
isFinite	isfinite	7.12.3.3

isZero	iszero	7.12.3.10
isSubnormal	issubnormal	7.12.3.9
isInfinite	isinf	7.12.3.4
isNaN	isnan	7.12.3.5
isSignaling	issignaling	7.12.3.8
isCanonical	iscanonical	7.12.3.2
radix	FLT_RADIX	5.2.4.2.2
totalOrder	totalorder	F.10.12.1
totalOrderMag	totalordermag	F.10.12.2
lowerFlags	feclearexcept	7.6.3.1
raiseFlags	fesetexcept	7.6.3.4
testFlags	fetestexcept	7.6.3.7
testSavedFlags	fetestexceptflag	7.6.3.6
restoreFlags	fesetexceptflag	7.6.3.5
saveAllFlags	fegetexceptflag	7.6.3.2
getBinaryRoundingDirection	fegetround	7.6.4.2
setBinaryRoundingDirection	fesetround	7.6.4.4
saveModes	fegetmode	7.6.4.1
restoreModes	fesetmode	7.6.4.3
defaultModes	fesetmode (FE_DFL_MODE)	7.6.4.3, 7.6

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

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

- 3 Whether *C* assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation³⁸¹) is implementation-defined, even if `<fenv.h>` defines the macro **FE_SNANS_ALWAYS_SIGNAL** (F.2.1). If the return expression of a **return** statement is evaluated to the floating-point format of the return type, it is implementation-defined whether a convertFormat operation is applied to the result of the return expression.
- 4 The unary **-** operator raises no floating-point exceptions, even if the operand is a signaling NaN.
- 5 The *C* classification macros **fpclassify**, **iscanonical**, **isfinite**, **isinf**, **isnan**, **isnormal**, **issignaling**, **issubnormal**, and **iszero** provide the IEC 60559 operations indicated in the table above provided their arguments are in the format of their semantic type. Then these macros raise no floating-point exceptions, even if an argument is a signaling NaN.
- 6 The *C* **nearbyint** functions (7.12.9.3, F.10.6.3) provide the nearbyinteger function recommended in the Appendix to (superseded) ANSI/IEEE 854.
- 7 The *C* **nextafter** (7.12.11.3, F.10.8.3) and **nexttoward** (7.12.11.4, F.10.8.4) functions provide the nextafter function recommended in the Appendix to (superseded) IEC 60559:1989 (but with a minor change to better handle signed zeros).
- 8 The *C* **getpayload**, **setpayload**, and **setpayloadsig** (F.10.13) functions provide program access to NaN payloads, defined in IEC 60559.
- 9 The macros (7.6) **FE_DOWNWARD**, **FE_TONEAREST**, **FE_TOWARDZERO**, and **FE_UPWARD**, which are used in conjunction with the **fegetround** and **fesetround** functions and the **FENV_ROUND** pragma, represent

³⁸¹)Where the source and destination formats are the same, convertFormat operations differ from copy operations in that convertFormat operations raise the “invalid” floating-point exception on signaling NaN inputs and do not propagate non-canonical encodings.

the IEC 60559 rounding-direction attributes `roundTowardNegative`, `roundTiesToEven`, `roundTowardZero`, and `roundTowardPositive`, respectively.

- 10 The C `fegetenv` (7.6.5.1), `feholdexcept` (7.6.5.2), `fesetenv` (7.6.5.3) and `feupdateenv` (7.6.5.4) functions provide a facility to manage the dynamic floating-point environment, comprising the IEC 60559 status flags and dynamic control modes.
- 11 IEC 60559 requires operations with specified operand and result formats. Therefore, math functions that are bound to IEC 60559 operations (see table above) must remove any extra range and precision from arguments or results.
- 12 IEC 60559 requires operations that round their result to formats the same as and wider than the operands, in addition to the operations that round their result to narrower formats (see 7.12.14). Operators (+, -, *, and /) whose evaluation formats are wider than the semantic type (5.2.4.2.2) might not support some of the IEEE 60559 operations, because getting a result in a given format might require a cast that could introduce an extra rounding error. The functions that round result to narrower type (7.12.14) provide the IEC 60559 operations that round result to same and wider (as well as narrower) formats, in those cases where built-in operators and casts do not. For example, `ddivl(x, y)` computes a correctly rounded `double` divide of `float x` by `float y`, regardless of the evaluation method.

F.4 Floating to integer conversion

- 1 If the integer type is `bool`, 6.3.1.2 applies and the conversion raises no floating-point exceptions if the floating-point value is not a signaling NaN. Otherwise, if the floating value is infinite or NaN or if the integral part of the floating value exceeds the range of the integer type, then the “invalid” floating-point exception is raised and the resulting value is unspecified. Otherwise, the resulting value is determined by 6.3.1.4. Conversion of an integral floating value that does not exceed the range of the integer type raises no floating-point exceptions; whether conversion of a non-integral floating value raises the “inexact” floating-point exception is unspecified.³⁸²⁾

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

- 1 Conversion from the widest supported IEC 60559 format to decimal with `DECIMAL_DIG` digits and back is the identity function.³⁸³⁾
- 2 Conversions involving IEC 60559 formats follow all pertinent recommended practice. In particular, conversion between any supported IEC 60559 format and decimal with `DECIMAL_DIG` or fewer significant digits is correctly rounded (honoring the current rounding mode), which assures that conversion from the widest supported IEC 60559 format to decimal with `DECIMAL_DIG` digits and back is the identity function.
- 3 The `<float.h>` header defines the macro

`CR_DECIMAL_DIG`

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

³⁸²⁾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`, `ufromfp`, `fromfpx`, `ufromfpx`, `rint`, `lrint`, `llrint`, and `nearbyint` in `<math.h>`.

³⁸³⁾If the minimum-width IEC 60559 binary64-extended format (64 bits of precision) is supported, `DECIMAL_DIG` is at least 21. If IEC 60559 binary64 (53 bits of precision) is the widest IEC 60559 format supported, then `DECIMAL_DIG` is at least 17. (By contrast, `LDBL_DIG` and `DBL_DIG` are 18 and 15, respectively, for these formats.)

F.10.6.6 The **round** functions

- 1 — **round**(± 0) returns ± 0 .
— **round**($\pm \infty$) returns $\pm \infty$.
- 2 The returned value is independent of the current rounding direction mode.
- 3 The **double** version of **round** behaves as though implemented by³⁹⁸⁾

```

#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double round(double x)
{
    double result;
    fenv_t save_env;
    feholdexcept(&save_env);
    result = rint(x);
    if (fetestexcept(FE_INEXACT)) {
        fesetround(FE_TOWARDZERO);
        result = rint(copysign(0.5 + fabs(x), x));
        feclearexcept(FE_INEXACT);
    }
    feupdateenv(&save_env);
    return result;
}

```

F.10.6.7 The **lround** and **llround** functions

- 1 The **lround** and **llround** functions differ from the **lrint** and **llrint** functions with the default rounding direction just in that the **lround** and **llround** functions round halfway cases away from zero and need not raise the “inexact” floating-point exception for non-integer arguments that round to within the range of the return type.

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 **fromfp** and **ufromfp** functions

- 1 The **fromfp** and **ufromfp** functions raise the “invalid” floating-point exception and return an unspecified value if the floating-point argument *x* is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width.
- 2 These functions do not raise the “inexact” floating-point exception.

³⁹⁸⁾This code does not handle signaling NaNs as required of implementations that define **FE_SNANS_ALWAYS_SIGNAL**.

J.5.11 Multiple external definitions

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

J.5.12 Predefined macro names

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

J.5.13 Floating-point status flags

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

J.5.14 Extra arguments for signal handlers

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

J.5.15 Additional stream types and file-opening modes

- 1 Additional mappings from files to streams are supported (7.21.2).
- 2 Additional file-opening modes may be specified by characters appended to the **mode** argument of the **fopen** function (7.21.5.3).

J.5.16 Defined file position indicator

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

J.5.17 Math error reporting

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

J.6 Reserved identifiers and keywords

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

J.6.1 Rule based identifiers

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

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

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

<code>_Alignas</code>	<code>atomic_load_explicit</code>
<code>__alignas_is_defined</code>	<code>atomic_long</code>
<code>_Alignof</code>	<code>ATOMIC_LONG_LOCK_FREE</code>
<code>__alignof_is_defined</code>	<code>ATOMIC_POINTER_LOCK_FREE</code>
<code>_Atomic</code>	<code>atomic_ptrdiff_t</code>
<code>atomic_bool</code>	<code>atomic_schar</code>
<code>ATOMIC_BOOL_LOCK_FREE</code>	<code>atomic_short</code>
<code>atomic_char</code>	<code>ATOMIC_SHORT_LOCK_FREE</code>
<code>atomic_char16_t</code>	<code>atomic_signal_fence</code>
<code>ATOMIC_CHAR16_T_LOCK_FREE</code>	<code>atomic_size_t</code>
<code>atomic_char32_t</code>	<code>atomic_store</code>
<code>ATOMIC_CHAR32_T_LOCK_FREE</code>	<code>atomic_store_explicit</code>
<code>ATOMIC_CHAR_LOCK_FREE</code>	<code>atomic_thread_fence</code>
<code>atomic_compare_exchange_strong</code>	<code>atomic_uchar</code>
<code>atomic_compare_exchange_strong_explicit</code>	<code>atomic_uint</code>
<code>atomic_compare_exchange_weak</code>	<code>atomic_uint_fast16_t</code>
<code>atomic_compare_exchange_weak_explicit</code>	<code>atomic_uint_fast32_t</code>
<code>atomic_exchange</code>	<code>atomic_uint_fast64_t</code>
<code>atomic_exchange_explicit</code>	<code>atomic_uint_fast8_t</code>
<code>atomic_fetch_</code>	<code>atomic_uint_least16_t</code>
<code>atomic_fetch_add</code>	<code>atomic_uint_least32_t</code>
<code>atomic_fetch_add_explicit</code>	<code>atomic_uint_least64_t</code>
<code>atomic_fetch_and</code>	<code>atomic_uint_least8_t</code>
<code>atomic_fetch_and_explicit</code>	<code>atomic_uintmax_t</code>
<code>atomic_fetch_or</code>	<code>atomic_uintptr_t</code>
<code>atomic_fetch_or_explicit</code>	<code>atomic_ullong</code>
<code>atomic_fetch_sub</code>	<code>atomic_ulong</code>
<code>atomic_fetch_sub_explicit</code>	<code>atomic_ushort</code>
<code>atomic_fetch_xor</code>	<code>ATOMIC_VAR_INIT</code>
<code>atomic_fetch_xor_explicit</code>	<code>atomic_wchar_t</code>
<code>atomic_flag</code>	<code>ATOMIC_WCHAR_T_LOCK_FREE</code>
<code>atomic_flag_clear</code>	<code>_Bool</code>
<code>atomic_flag_clear_explicit</code>	<code>__bool_true_false_are_defined</code>
<code>ATOMIC_FLAG_INIT</code>	<code>cond_broadcast</code>
<code>atomic_flag_test_and_set</code>	<code>cond_destroy</code>
<code>atomic_flag_test_and_set_explicit</code>	<code>cond_init</code>
<code>atomic_init</code>	<code>cond_signal</code>
<code>atomic_int</code>	<code>cond_t</code>
<code>atomic_int_fast16_t</code>	<code>cond_timedwait</code>
<code>atomic_int_fast32_t</code>	<code>cond_wait</code>
<code>atomic_int_fast64_t</code>	<code>_Complex</code>
<code>atomic_int_fast8_t</code>	<code>_Complex_I</code>
<code>atomic_int_least16_t</code>	<code>__cplusplus</code>
<code>atomic_int_least32_t</code>	<code>__DATE__</code>
<code>atomic_int_least64_t</code>	<code>EDOM</code>
<code>atomic_int_least8_t</code>	<code>EILSEQ</code>
<code>ATOMIC_INT_LOCK_FREE</code>	<code>EOF</code>
<code>atomic_intmax_t</code>	<code>EOL</code>
<code>atomic_intptr_t</code>	<code>ERANGE</code>
<code>atomic_is_lock_free</code>	<code>_Exit</code>
<code>atomic_llong</code>	<code>EXIT_FAILURE</code>
<code>ATOMIC_LLONG_LOCK_FREE</code>	<code>EXIT_SUCCESS</code>
<code>atomic_load</code>	<code>_EXT__</code>

FE_ALL_EXCEPT	_IOLBF
FE_DFL_ENV	_IONBF
FE_DFL_MODE	isalnum
FE_DIVBYZERO	isalpha
FE_DOWNWARD	isblank
FE_DYNAMIC	iscanonical
FE_INEXACT	iscntrl
FE_INVALID	isdigit
FE_OVERFLOW	iseqsig
FE_SNANS_ALWAYS_SIGNAL	isfinite
FE_TONEAREST	isgraph
FE_TOWARDZERO	isgreater
FE_UNDERFLOW	isgreaterequal
FE_UPWARD	isinf
__FILE__	isless
__func__	islessequal
_Generic	islessgreater
__Imaginary	islower
__Imaginary_I	isnan
INT16_C	isnormal
INT16_MAX	isprint
INT16_MIN	ispunct
int16_t	issignaling
INT32_C	isspace
INT32_MAX	issubnormal
INT32_MIN	isunordered
int32_t	isupper
INT64_C	iswalnum
INT64_MAX	iswalpha
INT64_MIN	iswblank
int64_t	iswcntrl
INT8_C	iswctype
INT8_MAX	iswdigit
INT8_MIN	iswgraph
int8_t	iswlower
int_fast16_t	iswprint
int_fast32_t	iswpunct
int_fast64_t	iswspace
int_fast8_t	iswupper
int_least16_t	iswxdigit
int_least32_t	isxdigit
int_least64_t	iszero
int_least8_t	LC_ALL
INT_MAX	LC_COLLATE
INTMAX_C	LC_CTYPE
INTMAX_MAX	LC_MONETARY
INTMAX_MIN	LC_NUMERIC
intmax_t	LC_TIME
INTMAX_WIDTH	__LINE__
INT_MIN	memchr
INTPTR_MAX	memcmp
INTPTR_MIN	memcpy
intptr_t	memcpy_s
INTPTR_WIDTH	memmove
INT_WIDTH	memmove_s
_IOFBF	memory_order

memory_order_acq_rel	PRIXLEAST32
memory_order_acquire	PRIXLEAST64
memory_order_consume	PRIXMAX
memory_order_relaxed	PRIXPTR
memory_order_release	SCNdMAX
memory_order_seq_cst	SCNdPTR
memset	SCNiMAX
memset_s	SCNiPTR
mtx_destroy	SCNoMAX
mtx_init	SCNoPTR
mtx_lock	SCNuMAX
mtx_plain	SCNuPTR
mtx_recursive	SCNxMAX
mtx_t	SCNxPTR
mtx_timed	SIGABRT
mtx_timedlock	SIG_ATOMIC_MAX
mtx_trylock	SIG_ATOMIC_MIN
mtx_unlock	SIG_ATOMIC_WIDTH
__Noreturn	SIG_DFL
__Pragma	SIG_ERR
PRId32	SIGFPE
PRId64	SIG_IGN
PRIdFAST32	SIGILL
PRIdFAST64	SIGINT
PRIdLEAST32	SIGSEGV
PRIdLEAST64	SIGTERM
PRIdMAX	__Static_assert
PRIdPTR	__STDC__
PRi32	__STDC_ANALYZABLE__
PRi64	__STDC_HOSTED__
PRiFAST32	__STDC_IEC_559__
PRiFAST64	__STDC_IEC_559_COMPLEX__
PRiLEAST32	__STDC_IEC_60559_BFP__
PRiLEAST64	__STDC_IEC_60559_COMPLEX__
PRiMAX	__STDC_ISO_10646__
PRiPTR	__STDC_LIB_EXT1__
PRIo32	__STDC_MB_MIGHT_NEQ_WC__
PRIo64	__STDC_NO_ATOMICS__
PRIoFAST32	__STDC_NO_COMPLEX__
PRIoFAST64	__STDC_NO_THREADS__
PRIoLEAST32	__STDC_NO_VLA__
PRIoLEAST64	__STDC_UTF_16__
PRIoMAX	__STDC_UTF_32__
PRIoPTR	__STDC_VERSION__
PRiU32	__STDC_WANT_IEC_60559__
PRiU64	__STDC_WANT_IEC_60559_BFP_EXT__
PRiUFAST32	__STDC_WANT_LIB_EXT1__
PRiUFAST64	
PRiULEAST32	strcat
PRiULEAST64	strcat_s
PRiUMAX	strchr
PRiUPTR	strcmp
PRIX32	strcoll
PRIX64	strcpy
PRIXFAST32	strcpy_s
PRIXFAST64	strcspn
	strerror

strerrorlen_s	towctrans
strerror_s	towlower
strfromd	toupper
strfromf	tss_create
strfroml	tss_delete
strftime	tss_dtor_t
strlen	tss_get
strncat	tss_set
strncat_s	tss_t
strncmp	UINT16_C
strncpy	UINT16_MAX
strncpy_s	uint16_t
strnlen_s	UINT32_C
strpbrk	UINT32_MAX
strrchr	uint32_t
strspn	UINT64_C
strstr	UINT64_MAX
strtod	uint64_t
strtof	UINT8_C
strtoimax	UINT8_MAX
strtok	uint8_t
strtok_s	uint_fast16_t
strtol	uint_fast32_t
strtold	uint_fast64_t
strtoll	uint_fast8_t
strtoul	uint_least16_t
strtoull	uint_least32_t
strtoumax	uint_least64_t
struct	uint_least8_t
strxfrm	UINT_MAX
thrd_busy	UINTMAX_C
thrd_create	UINTMAX_MAX
thrd_current	uintmax_t
thrd_detach	UINTMAX_WIDTH
thrd_equal	UINTPTR_MAX
thrd_error	uintptr_t
thrd_exit	UINTPTR_WIDTH
thrd_join	UINT_WIDTH
thrd_nOMEM	__VA_ARGS__
thrd_sleep	wscat
thrd_start_t	wscat_s
thrd_success	wcschr
thrd_t	wscmp
thrd_timedout	wscoll
thrd_yield	wscpy
_Thread_local	wscpy_s
__TIME__	wscspn
TIME_UTC	wcsftime
tolower	wcslen
totalorder	wcsncat
totalorderf	wcsncat_s
totalorderl	wcsncmp
totalordermag	wcsncpy
totalordermagf	wcsncpy_s
totalordermagl	wcsnlen_s
toupper	wcspbrk

wcsrchr	wcstol
wcsrtombs	wcstold
wcsrtombs_s	wcstoll
wcsspn	wcstombs
wcsstr	wcstombs_s
wcstod	wcstoul
wcstof	wcstoull
wcstoimax	wcstoumax
wcstok	wcsxfrm
wcstok_s	_WIDTH

J.6.2 Particular identifiers or keywords

- 1 The following ~~808~~809 identifiers or keywords are not covered by the above and have particular semantics provided by this document.

abort	bitor	cbrtf
abort_handler_s	bool	cbrtl
abs	break	ccos
acos	bsearch	ccosf
acosf	bsearch_s	ccosh
acosh	btowc	ccoshf
acoshf	BUFSIZ	ccoshl
acoshl	c16rtomb	ccosl
acosl	c32rtomb	ceil
alignas	cabs	ceilf
aligned_alloc	cabsf	ceill
alignof	cabsl	cerf
and	cacos	cerfc
and_eq	cacosf	cexp
asctime	cacosh	cexp2
asctime_s	cacoshf	cexpf
asin	cacoshl	cexpl
asinf	cacosl	cexpm1
asinh	calloc	char
asinhf	call_once	char16_t
asinhl	canonicalize	char32_t
asinl	canonicalizef	CHAR_BIT
assert	canonicalizel	CHAR_MAX
atan	carg	CHAR_MIN
atan2	cargf	CHAR_WIDTH
atan2f	cargl	cimag
atan2l	case	cimagf
atanf	casin	cimagl
atanh	casinf	clearerr
atanhf	casinh	clgamma
atanhl	casinhf	clock
atanl	casinhl	CLOCKS_PER_SEC
atexit	casinl	clock_t
atof	catan	clog
atoi	catanf	clog10
atol	catanh	clog1p
atoll	catanhf	clog2
at_quick_exit	catanhhl	clogf
auto	catanhl	clogl
bitand	catanl	CPLX
	cbrt	

CMPLEXF	DBL_MAX	faddl
CMPXLX	DBL_MAX_10_EXP	false
compl	DBL_MAX_EXP	fclose
complex	DBL_MIN	fdim
conj	DBL_MIN_10_EXP	fdimf
conjf	DBL_MIN_EXP	fdiml
conjl	DBL_TRUE_MIN	fdiv
const	ddiv	fdivl
constraint_handler_t	ddivl	feclearexcept
continue	DECIMAL_DIG	fegetenv
copysign	decimal_point	fegetexceptflag
copysignf	DEFAULT	fegetmode
copysignl	define	fegetround
cos	defined	feholdexcept
cosf	dfma	femode_t
cosh	dfmal	FENV_ACCESS
coshf	difftime	FENV_ROUND
coshl	div	fenv_t
cosl	div_t	feof
cpow	dmul	feraiseexcept
cpowf	dnull	ferror
cpowl	do	fesetenv
cproj	double	fesetexcept
cprojf	double_t	fesetexceptflag
cprojl	dsqrt	fesetmode
CR_DECIMAL_DIG	dsqrtl	fesetround
creal	dsub	fetestexcept
crealf	dsubl	fetestexceptflag
creall	elif	feupdateenv
csin	else	fexcept_t
csinf	endif	fflush
csinh	enum	ffma
csinhf	erf	ffmal
csinhl	erfc	fgetc
csinl	erfcf	fgetpos
csqrt	erfcl	fgets
csqrtf	erfff	fgetwc
csqrtl	erfl	fgetws
ctan	errno	FILE
ctanf	errno_t	FILENAME_MAX
ctanh	error	float
ctanhf	exit	float_t
ctanhl	exp	floor
ctanl	exp2	floorf
ctgamma	exp2f	floorl
ctime	exp2l	FLT_DECIMAL_DIG
ctime_s	expf	FLT_DIG
currency_symbol	expl	FLT_EPSILON
CX_LIMITED_RANGE	expl	FLT_EVAL_METHOD
dadd	expm1	FLT_HAS_SUBNORM
daddl	expm1f	FLT_MANT_DIG
DBL_DECIMAL_DIG	expm1l	FLT_MAX
DBL_DIG	extern	FLT_MAX_10_EXP
DBL_EPSILON	fabs	FLT_MAX_EXP
DBL_HAS_SUBNORM	fabsf	FLT_MIN
DBL_MANT_DIG	fabsl	FLT_MIN_10_EXP
	fadd	

FLT_MIN_EXP	FP_INT_TONEARESTFROMZERO	gmtime
FLT_RADIX	FP_INT_TOWARDZERO	gmtime_s
FLT_ROUNDS	FP_INT_UPWARD	goto
FLT_TRUE_MIN	FP_LLOGB0	grouping
fma	FP_LLOGBNAN	HUGE_VAL
fmaf	FP_NAN	HUGE_VALF
fmal	FP_NORMAL	HUGE_VALL
fmax	fpos_t	hypot
fmaxf	fprintf	hypotf
fmaxl	fprintf_s	hypotl
fmaxmag	FP_SUBNORMAL	I
fmaxmagf	fputc	if
fmaxmagl	fputs	ifdef
fmin	fputwc	ifndef
fminf	fputws	ignore_handler_s
fminl	FP_ZERO	ilogb
fminmag	frac_digits	ilogbf
fminmagf	fread	ilogbl
fminmagl	free	imaginary
fmod	freopen	imaxabs
fmodf	freopen_s	imaxdiv
fmodl	frexp	imaxdiv_t
fmul	frexpf	include
fmul_l	frexpl	INFINITY
fopen	fromfp	inline
FOPEN_MAX	fromfpf	int_curr_symbol
fopen_s	fromfp_l	int_frac_digits
for	fromfp_x	int_n_cs_precedes
fpclassify	fromfp_xf	int_n_sep_by_space
FP_CONTRACT	fromfp_xl	int_n_sign_posn
FP_FAST_DADDL	fscanf	int_p_cs_precedes
FP_FAST_DDIVL	fscanf_s	int_p_sep_by_space
FP_FAST_DFMAL	fseek	int_p_sign_posn
FP_FAST_DMULL	fsetpos	jmp_buf
FP_FAST_DSQRTL	fsqrt	kill_dependency
FP_FAST_DSUBL	fsqrt_l	labs
FP_FAST_FADD	fsub	lconv
FP_FAST_FADDL	fsub_l	LDBL_DECIMAL_DIG
FP_FAST_FDIV	ftell	LDBL_DIG
FP_FAST_FDIVL	fwide	LDBL_EPSILON
FP_FAST_FFMA	fwprintf	LDBL_HAS_SUBNORM
FP_FAST_FFMAL	fwprintf_s	LDBL_MANT_DIG
FP_FAST_FMA	fwrite	LDBL_MAX
FP_FAST_FMAF	fwscanf	LDBL_MAX_10_EXP
FP_FAST_FMAL	fwscanf_s	LDBL_MAX_EXP
FP_FAST_FMUL	getc	LDBL_MIN
FP_FAST_FMULL	getchar	LDBL_MIN_10_EXP
FP_FAST_FSQRT	getenv	LDBL_MIN_EXP
FP_FAST_FSQRTL	getenv_s	LDBL_TRUE_MIN
FP_FAST_FSUB	getpayload	ldexp
FP_FAST_FSUBL	getpayloadf	ldexpf
FP_ILOGB0	getpayload_l	ldexpl
FP_ILOGBNAN	gets	ldiv
FP_INFINITE	gets_s	ldiv_t
FP_INT_DOWNWARD	getwc	lgamma
FP_INT_TONEAREST	getwchar	lgammaf

lgammal	MB_LEN_MAX	positive_sign
line	mbrlen	pow
llabs	mbrtoc16	powf
lldiv	mbrtoc32	powl
lldiv_t	mbrtowc	pragma
llogb	mbsinit	printf
llogbf	mbsrtowcs	printf_s
llogbl	mbsrtowcs_s	p_sep_by_space
LLONG_MAX	mbstate_t	p_sign_posn
LLONG_MIN	mbstowcs	PTRDIFF_MAX
LLONG_WIDTH	mbstowcs_s	PTRDIFF_MIN
llrint	mbtowc	ptrdiff_t
llrintf	mktime	PTRDIFF_WIDTH
llrintl	modf	putc
llround	modff	putchar
llroundf	modfl	puts
llroundl	mon_decimal_point	putwc
localeconv	mon_grouping	putwchar
localtime	mon_thousands_sep	qsort
localtime_s	nan	qsort_s
log	nanf	quick_exit
log10	nanl	raise
log10f	n_cs_precedes	rand
log10l	NDEBUG	RAND_MAX
log1p	nearbyint	realloc
log1pf	nearbyintf	register
log1pl	nearbyintl	remainder
log2	negative_sign	remainderf
log2f	nextafter	remainderl
log2l	nextafterf	remove
logb	nextafterl	remquo
logbf	nextdown	remquoof
logbl	nextdownf	remquoof
logf	nextdownl	rename
logl	nexttoward	restrict
long	nexttowardf	return
longjmp	nexttowardl	rewind
LONG_MAX	nextup	rint
LONG_MIN	nextupf	rintf
LONG_WIDTH	nextupl	rintl
lrint	noreturn	round
lrintf	not	roundeven
lrintl	not_eq	roundevenf
lround	n_sep_by_space	roundevenl
lroundf	n_sign_posn	roundf
lroundl	NULL	roundl
L_tmpnam	nullptr	RSIZE_MAX
L_tmpnam_s	OFF	rsize_t
main	offsetof	scalbn
malloc	ON	scalbnf
MATH_ERREXCEPT	once_flag	scalbnl
math_errhandling	ONCE_FLAG_INIT	scalbn
MATH_ERRNO	or	scalbnf
max_align_t	or_eq	scalbnl
MB_CUR_MAX	p_cs_precedes	scanf
mblen	perror	scanf_s

WINT_MAX
WINT_MIN
wint_t
WINT_WIDTH
wmemchr
wmemcmp

wmemcpy
wmemcpy_s
wmemmove
wmemmove_s
wmemset
wprintf

wprintf_s
wscanf
wscanf_s
xor
xor_eq

Annex K

(normative)

Bounds-checking interfaces

K.1 Background

- 1 Traditionally, the C Library has contained many functions that trust the programmer to provide output character arrays big enough to hold the result being produced. Not only do these functions not check that the arrays are big enough, they frequently lack the information needed to perform such checks. While it is possible to write safe, robust, and error-free code using the existing library, the library tends to promote programming styles that lead to mysterious failures if a result is too big for the provided array.
- 2 A common programming style is to declare character arrays large enough to handle most practical cases. However, if these arrays are not large enough to handle the resulting strings, data can be written past the end of the array overwriting other data and program structures. The program never gets any indication that a problem exists, and so never has a chance to recover or to fail gracefully.
- 3 Worse, this style of programming has compromised the security of computers and networks. Buffer overflows can often be exploited to run arbitrary code with the permissions of the vulnerable (defective) program.
- 4 If the programmer writes runtime checks to verify lengths before calling library functions, then those runtime checks frequently duplicate work done inside the library functions, which discover string lengths as a side effect of doing their job.
- 5 This annex provides alternative library functions that promote safer, more secure programming. The alternative functions verify that output buffers are large enough for the intended result and return a failure indicator if they are not. Data is never written past the end of an array. All string results are null terminated.
- 6 This annex also addresses another problem that complicates writing robust code: functions that are not reentrant because they return pointers to static objects owned by the function. Such functions can be troublesome since a previously returned result can change if the function is called again, perhaps by another thread.

K.2 Scope

- 1 This annex specifies a series of optional extensions that can be useful in the mitigation of security vulnerabilities in programs, and comprise new functions, macros, and types declared or defined in existing standard headers.
- 2 An implementation that defines `__STDC_LIB_EXT1__` shall conform to the specifications in this annex.⁴⁰⁶⁾
- 3 Subclause K.3 should be read as if it were merged into the parallel structure of named subclauses of Clause 7.

K.3 Library

K.3.1 Introduction

K.3.1.1 Standard headers

- 1 The functions, macros, and types declared or defined in K.3 and its subclauses are not declared or defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is defined as a macro which expands to the integer constant 0 at the point in the source file where the appropriate header is first included.
- 2 The functions, macros, and types declared or defined in K.3 and its subclauses are declared and defined by their respective headers if `__STDC_WANT_LIB_EXT1__` is defined as a macro which expands to the integer constant 1 at the point in the source file where the appropriate header is first

⁴⁰⁶⁾Implementations that do not define `__STDC_LIB_EXT1__` are not required to conform to these specifications.