WG 14 N2579

Annex X
(normative)
IEC 60559 interchange and extended types

X.1 Introduction

[1] This annex specifies extension types for programming language C that have the arithmetic interchange and extended floating-point formats specified in ISO/IEC/IEEE 60559. This annex also includes functions that support the non-arithmetic interchange formats in that standard. This annex was adapted from ISO/IEC TS 18661-3:2015, *Floating-point extensions for C —Interchange and extended types*.

[2] An implementation that defines **`__STDC_IEC_60559_TYPES__`** to **`20yymmL`** shall conform to the specifications in this annex. An implementation may define **`__STDC_IEC_60559_TYPES__`** only if it defines **`__STDC_IEC_60559_BFP__`**, indicating support for IEC 60559 binary floating-point arithmetic, or defines **`__STDC_IEC_60559_DFP__`**, indicating support for IEC 60559 decimal floating-point arithmetic (or defines both). Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

=========================

Change to C2X working draft N2478:

In 6.10.8.3#1, add:

`__STDC_IEC_60559_TYPES__` The integer constant **`20yymmL`**, intended to indicate conformance to the specification in Annex X (IEC 60559 interchange and extended types).

=========================

X.2 Types

[1] This clause specifies types that support IEC 60559 arithmetic interchange and extended formats. The encoding conversion functions (X.11.3) and numeric conversion functions for encodings (X.12.3, X.12.4) support the non-arithmetic interchange formats specified in IEC 60559.

X.2.1 Interchange floating types

[1] IEC 60559 specifies interchange formats, and their encodings, which can be used for the exchange of floating-point data between implementations. These formats are identified by their radix (binary or decimal) and their storage width *N*. The two tables below give the C floating-point model parameters*) (5.2.4.2.2) for the IEC 60559 interchange formats, where the function round() rounds to the nearest integer.

Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary*N* ($N \geq 128$)
N, storage width in bits	16	32	64	128	*N* a multiple of 32
p, precision in bits	11	24	53	113	$N - \text{round}(4 \times \log_2(N)) + 13$
emax, maximum exponent *e*	16	128	1024	16384	$2^{(N-p-1)}$
emin, minimum exponent *e*	−13	−125	−1021	−16381	3 − *emax*

Decimal interchange format parameters

Parameter	decimal32	decimal64	decimal128	decimal*N* ($N \geq 32$)
N, storage width in bits	32	64	128	*N* a multiple of 32
p, precision in digits	7	16	34	$9 \times N/32 - 2$
emax, maximum exponent *e*	97	385	6145	$3 \times 2^{(N/16\,+\,3)} + 1$
emin, minimum exponent *e*	−94	−382	−6142	3 − *emax*

*) In IEC 60559, normal floating-point numbers are expressed with the first significant digit to the left of the radix point. Hence the exponent in the C model (shown in the tables) is 1 more than the exponent of the same number in the IEC 60559 model.

[2] **EXAMPLE** For the binary160 format, *p* = 144, *emax* = 32768 and *emin* = −32765. For the decimal160 format, *p* = 43, *emax* = 24577, and *emin* = −24574.

[3] Types designated

`_Float`*N*, where *N* is 16, 32, 64, or ≥ 128 and a multiple of 32

and types designated

`_Decimal`*N*, where *N* ≥ 32 and a multiple of 32

are collectively called the *interchange floating types*. Each interchange floating type has the IEC 60559 interchange format corresponding to its width (*N*) and radix (2 for `_Float`*N*, 10 for `_Decimal`*N*). Each interchange floating type is not compatible with any other type.

[4] An implementation that defines `__STDC_IEC_60559_BFP__` and `__STDC_IEC_60559_TYPES__` shall provide `_Float32` and `_Float64` as interchange floating types with the same representation and alignment requirements as `float` and `double`, respectively. If the implementation's `long double` type supports an IEC 60559 interchange format of width *N* > 64, then the implementation shall also provide the type `_Float`*N* as an interchange floating type with the same representation and alignment requirements as `long double`. The implementation may provide other radix-2 interchange floating types `_Float`*N*; the set of such types supported is implementation-defined.

[5] An implementation that defines `__STDC_IEC_60559_DFP__` provides the decimal floating types `_Decimal32`, `_Decimal64`, and `_Decimal128`. (6.2.5). If the implementation also defines

__STDC_IEC_60559_TYPES__, it may provide other radix-10 interchange floating types **_Decimal***N*; the set of such types supported is implementation-defined.

X.2.2 Non-arithmetic interchange formats

[1] An implementation supports IEC 60559 non-arithmetic interchange formats by providing the associated encoding-to-encoding conversion functions (X.11.3.2) in **<math.h>** and the string-from-encoding functions (X.12.3) and string-to-encoding functions (X.12.4) in **<stdlib.h>**.

[2] An implementation that defines **__STDC_IEC_60559_BFP__** and **__STDC_IEC_60559_TYPES__** supports some IEC 60559 radix-2 interchange formats as arithmetic formats by providing types **_Float***N* (as well as **float** and **double**) with those formats. The implementation may support other IEC 60559 radix-2 interchange formats as non-arithmetic formats; the set of such formats supported is implementation-defined.

[3] An implementation that defines **__STDC_IEC_60559_DFP__** and **__STDC_IEC_60559_TYPES__** supports some IEC 60559 radix-10 interchange formats as arithmetic formats by providing types **_Decimal***N* with those formats. The implementations may support other IEC 60559 radix-10 interchange formats as non-arithmetic formats; the set of such formats supported is implementation-defined.

X.2.3 Extended floating types

[1] For each of its basic formats, IEC 60559 specifies an extended format whose maximum exponent and precision exceed those of the basic format it is associated with. Extended formats are intended for arithmetic with more precision and exponent range than is available in the basic formats used for the input data. The extra precision and range often mitigate round-off error and eliminate overflow and underflow in intermediate computations. The table below gives the minimum values of these parameters, as defined for the C floating-point model (5.2.4.2.2). For all IEC 60559 extended (and interchange) formats, *emin* = 3 − *emax*.

Extended format parameters for floating-point numbers

	Extended formats associated with:				
Parameter	binary32	binary64	binary128	decimal64	decimal128
p digits ≥	32	64	128	22	40
emax ≥	1024	16384	65536	6145	24577

[2] Types designated **_Float32x**, **_Float64x**, **_Float128x**, **_Decimal64x**, and **_Decimal128x** support the corresponding IEC 60559 extended formats and are collectively called the *extended floating types*. The set of values of **_Float32x** is a subset of the set of values of **_Float64x**; the set of values of **_Float64x** is a subset of the set of values of **_Float128x**. The set of values of **_Decimal64x** is a subset of the set of values of **_Decimal128x**. Each extended floating type is not compatible with any other type. An implementation that defines **__STDC_IEC_60559_BFP__** and **__STDC_IEC_60559_TYPES__** shall provide **_Float32x**, and may provide one or both of the types **_Float64x** and **_Float128x.** An implementation that defines **__STDC_IEC_60559_DFP__** and **__STDC_IEC_60559_TYPES__** shall provide **_Decimal64x**, and may provide **_Decimal128x**. Which (if any) of the optional extended floating types are provided is implementation-defined.

[3] **NOTE** IEC 60559 does not specify an extended format associated with the decimal32 format, nor does this annex specify an extended type associated with the **_Decimal32** type.

[4] **NOTE** The **_Float32x** type may have the same format as **double**. The **_Decimal64x** type may have the same format as **_Decimal128**.

X.2.4 Classification of real floating types

[1] 6.2.5 defines standard floating types as a collective name for the types **`float`**, **`double`**, and **`long double`** and it defines decimal floating types as a collective name for the types **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`**.

[2] X.2.1 defines interchange floating types and X.2.3 defines extended floating types.

[3] The types **`_Float`***N* and **`_Float`***N***`x`** are collectively called *binary floating types*.

[4] This subclause broadens *decimal floating types* to include the types **`_Decimal`***N* and **`_Decimal`***N***`x`** introduced in this annex, as well as **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`**.

[5] This subclause broadens *real floating types* to include all interchange floating types and extended floating types, as well as standard floating types.

[6] Thus, in this annex, real floating types are classified as follows:

standard floating types:
  `float`
  `double`
  `long double`

decimal floating types:
  `_Decimal`*N*
  `_Decimal`*N***`x`**

binary floating types:
  `_Float`*N*
  `_Float`*N***`x`**

interchange floating types:
  `_Float`*N*
  `_Decimal`*N*

extended floating types:
  `_Float`*N***`x`**
  `_Decimal`*N***`x`**

[7] **NOTE** Standard floating types (which have an implementation-defined radix) are not included in either binary floating types (which always have radix 2) or decimal floating types (which always have radix 10).

X.2.5 Complex types

[1] This subclause broadens the C complex types (6.2.5) to also include similar types whose corresponding real parts have binary floating types. For the types **`_Float`***N* and **`_Float`***N***`x`**, there are complex types designated respectively as **`_Float`***N* **`_Complex`** and **`_Float`***N***`x`** **`_Complex`**. (Complex types are a conditional feature that implementations need not support; see 6.10.8.3.)

X.2.6 Imaginary types

[1] This subclause broadens the C imaginary types (G.2) to also include similar types whose corresponding real parts have binary floating types. For the types **`_Float`***N* and **`_Float`***N***`x`**, there are imaginary types designated respectively as **`_Float`***N* **`_Imaginary`** and **`_Float`***N***`x`** **`_Imaginary`**. The imaginary types (along with the real floating and complex types) are floating types. (Annex G,

including imaginary types, is a conditional feature that implementations need not support; see 6.10.8.3.)

X.3 Characteristics in <float.h>

[1] This subclause enhances the **FLT_EVAL_METHOD** and **DEC_EVAL_METHOD** macros to apply to the types introduced in this annex.

[2] If **FLT_RADIX** is 2, the value of the macro **FLT_EVAL_METHOD** (5.2.4.2.2) characterizes the use of evaluation formats for standard floating types and for binary floating types:

−1 indeterminable;

0 evaluate all operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **float**, to the range and precision of **float**; evaluate all other operations and constants to the range and precision of the semantic type;

1 evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **double**, to the range and precision of **double**; evaluate all other operations and constants to the range and precision of the semantic type;

2 evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **long double**, to the range and precision of **long double**; evaluate all other operations and constants to the range and precision of the semantic type;

N, where **_Float***N* is a supported interchange floating type
evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **_Float***N*, to the range and precision of **_Float***N*; evaluate all other operations and constants to the range and precision of the semantic type;

N + 1, where **_Float***N***x** is a supported extended floating type
evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **_Float***N***x**, to the range and precision of **_Float***N***x**; evaluate all other operations and constants to the range and precision of the semantic type.

If **FLT_RADIX** is not 2, the use of evaluation formats for operations and constants of binary floating types is implementation-defined.

[3] The implementation-defined value of the macro **DEC_EVAL_METHOD** (5.2.4.2.3) characterizes the use of evaluation formats for decimal floating types:

−1 indeterminable;

0 evaluate all operations and constants just to the range and precision of the type;

1 evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **_Decimal64**, to the range and precision of **_Decimal64**; evaluate all other operations and constants to the range and precision of the semantic type;

2 evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of **_Decimal128**, to the range and precision of

`_Decimal128`; evaluate all other operations and constants to the range and precision of the semantic type;

N, where `_Decimal`*N* is a supported interchange floating type
: evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `_Decimal`*N*, to the range and precision of `_Decimal`*N*; evaluate all other operations and constants to the range and precision of the semantic type;

N + 1, where `_Decimal`*N*`x` is a supported extended floating type
: evaluate operations and constants, whose semantic type comprises a set of values that is a strict subset of the values of `_Decimal`*N*`x`, to the range and precision of `_Decimal`*N*`x`; evaluate all other operations and constants to the range and precision of the semantic type.

[4] This subclause also specifies `<float.h>` macros, analogous to the macros for standard floating types, that characterize binary floating types in terms of the model presented in 5.2.4.2.2. This subclause generalizes the specification of characteristics in 5.2.4.2.3 to include the decimal floating types introduced in this annex. The prefix `FLT`*N*`_` indicates the type `_Float`*N* or the non-arithmetic binary interchange format of width *N*. The prefix `FLT`*N*`X_` indicates the type `_Float`*N*`x`. The prefix `DEC`*N*`_` indicates the type `_Decimal`*N* or the non-arithmetic decimal interchange format of width *N*. The prefix `DEC`*N*`X_` indicates the type `_Decimal`*N*`x`. The type parameters p, e_{max}, and e_{min} for extended floating types are for the extended floating type itself, not for the basic format that it extends.

[5] If `__STDC_WANT_IEC_60559_TYPES_EXT__` is defined (by the user) at the point in the code where `<float.h>` is first included, the following applies (X.8). For each interchange or extended floating type that the implementation provides, `<float.h>` shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, `<float.h>` shall not define the associated macros in the following list, except, the implementation shall define the macros `FLT`*N*`_DECIMAL_DIG` and `FLT`*N*`_DIG` if it supports the IEC 60559 non-arithmetic binary interchange format of width *N* (X.2.2).

[6] The integer values given in the following lists shall be replaced by constant expressions suitable for use in `#if` preprocessing directives:

— radix of exponent representation, b (= 2 for binary, 10 for decimal)

For the standard floating types, this value is implementation-defined and is specified by the macro `FLT_RADIX`. For the interchange and extended floating types there is no corresponding macro; the radix is an inherent property of the types.

— number of bits in the floating-point significand, p

`FLT`*N*`_MANT_DIG`
`FLT`*N*`X_MANT_DIG`

— number of digits in the coefficient, p

`DEC`*N*`_MANT_DIG`
`DEC`*N*`X_MANT_DIG`

— number of decimal digits, n, such that any floating-point number with p bits can be rounded to a floating-point number with n decimal digits and back again without change to the value, $\lceil 1 + p \log_{10} 2 \rceil$

FLT*N*_DECIMAL_DIG
FLT*N*X_DECIMAL_DIG

— number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p bits and back again without change to the q decimal digits, $\lfloor (p - 1) \log_{10} 2 \rfloor$

FLT*N*_DIG
FLT*N*X_DIG

— minimum negative integer such that the radix raised to one less than that power is a normalized floating-point number, e_{min}

FLT*N*_MIN_EXP
FLT*N*X_MIN_EXP
DEC*N*_MIN_EXP
DEC*N*X_MIN_EXP

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} 2^{emin-1} \rceil$

FLT*N*_MIN_10_EXP
FLT*N*X_MIN_10_EXP

— maximum integer such that the radix raised to one less than that power is a representable finite floating-point number, e_{max}

FLT*N*_MAX_EXP
FLT*N*X_MAX_EXP
DEC*N*_MAX_EXP
DEC*N*X_MAX_EXP

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - 2^{-p})2^{\mathrm{emax}}) \rfloor$

FLT*N*_MAX_10_EXP
FLT*N*X_MAX_10_EXP

— maximum representable finite floating-point number, $(1 - b^{-p})b^{\mathrm{emax}}$

FLT*N*_MAX
FLT*N*X_MAX
DEC*N*_MAX
DEC*N*X_MAX

— the difference between 1 and the least value greater than 1 that is representable in the given floating-point type, b^{1-p}

FLT*N*_EPSILON
FLT*N*X_EPSILON
DEC*N*_EPSILON
DEC*N*X_EPSILON

— minimum normalized positive floating-point number, b^{emin-1}

FLT*N*_MIN
FLT*N*X_MIN
DEC*N*_MIN
DEC*N*X_MIN

— minimum positive floating-point number, b^{emin-p}

FLT*N*_TRUE_MIN
FLT*N*X_TRUE_MIN
DEC*N*_TRUE_MIN
DEC*N*X_TRUE_MIN

X.4 Conversions

[1] This subclause enhances the usual arithmetic conversions (6.3.1.8) to handle interchange and extended floating types. It supports the IEC 60559 recommendation against allowing implicit conversions of operands to obtain a common type where the conversion is between types where neither is a subset of (or equivalent to) the other.

[2] This subclause also broadens the operation binding in F.3 for the IEC 60559 convertFormat operation to apply to IEC 60559 arithmetic and non-arithmetic formats

X.4.1 Real floating and integer

[1] When a finite value of interchange or extended floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the "invalid" floating-point exception shall be raised and the result of the conversion is unspecified.

[2] When a value of integer type is converted to an interchange or extended floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted cannot be represented exactly, the result shall be correctly rounded with exceptions raised as specified in IEC 60559.

X.4.2 Usual arithmetic conversions

[1] If either operand is of floating type, the common real type is determined as follows:

If one operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

If only one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

If both operands have the same corresponding real type, no further conversion is needed.

If both operands have floating types and neither of the sets of values of their corresponding real types is a subset of (or equivalent to) the other, the behavior is undefined.

Otherwise, if both operands are floating types and the sets of values of their corresponding real types are not equivalent, the operand whose set of values of its corresponding real type is a strict subset of the set of values of the corresponding real type of the other operand is converted, without change of type domain, to a type with the corresponding real type of that other operand.

Otherwise, if both operands are floating types and the sets of values of their corresponding real types are equivalent, then the following rules are applied:

If the corresponding real type of either operand is an interchange floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same interchange floating type.

Otherwise, if the corresponding real type of either operand is **`long double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`long double`**.

Otherwise, if the corresponding real type of either operand is **`double`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`double`**.

(All cases where **`float`** might have the same format as another type are covered above.)

Otherwise, if the corresponding real type of either operand is **`_Float128x`** or **`_Decimal128x`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`_Float128x`** or **`_Decimal128x`**, respectively.

Otherwise, if the corresponding real type of either operand is **`_Float64x`** or **`_Decimal64x`**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **`_Float64x`** or **`_Decimal64x`**, respectively.

X.4.3 Arithmetic and non-arithmetic formats

[1] The operation binding in F.3 for the IEC 60559 convertFormat operation applies to IEC 60559 arithmetic and non-arithmetic formats as follows:

— For conversions between arithmetic formats supported by floating types (same or different radix) - casts and implicit conversions.

— For same-radix conversions between non-arithmetic interchange formats - encoding-to-encoding conversion functions (X.11.3.2).

— For conversions between non-arithmetic interchange formats (same or different radix) – compositions of string-from-encoding functions (X.12.3) (converting exactly) and string-to-encoding functions (X.12.4).

— For same-radix conversions from interchange formats supported by interchange floating types to non-arithmetic interchange formats – compositions of encode functions (X.11.3.1.1, 7.12.16.1, 7.12.16.3) and encoding-to-encoding functions (X.11.3.2).

— For same radix conversions from non-arithmetic interchange formats to interchange formats supported by interchange floating types – compositions of encoding-to-encoding conversion functions (X.11.3.2) and decode functions (X.11.3.1.2, 7.12.16.2, 7.12.16.4). See the example in X.11.3.2.1.

— For conversions from non-arithmetic interchange formats to arithmetic formats supported by floating types (same or different radix) – compositions of string-from-encoding functions (X.12.3) (converting exactly) and numeric conversion functions **`strtod`**, etc. (7.22.1.5, 7.22.1.6). See the example in X.12.2.

— For conversions from arithmetic formats supported by floating types to non-arithmetic interchange formats (same or different radix) – compositions of numeric conversion functions **strfromd**, etc. (7.22.1.3, 7.22.1.4) (converting exactly) and string-to-encoding functions (X.12.4).

X.5 Lexical elements

X.5.1 Keywords

[1] This subclause expands the list of keywords (6.4.1) to also include:

_Float*N*, where *N* is 16, 32, 64, or ≥ 128 and a multiple of 32
_Float32x
_Float64x
_Float128x
_Decimal*N*, where *N* is 96 or > 128 and a multiple of 32
_Decimal64x
_Decimal128x

X.5.2 Constants

[1] This subclause specifies constants of interchange and extended floating types.

[2] This subclause expands *floating-suffix* (6.4.4.2) to also include:

f*N* **F***N* **f***N***x** **F***N***x** **d***N* **D***N* **d***N***x** **D***N***x**

[3] A floating suffix **d***N*, **D***N*, **d***N***x**, or **D***N***x** shall not be used in a *hexadecimal-floating-constant*.

[4] A floating suffix shall not designate a type that the implementation does not provide.

[5] If a floating constant is suffixed by **f***N* or **F***N*, it has type **_Float***N*. If suffixed by **f***N***x** or **F***N***x**, it has type **_Float***N***x**. If suffixed by **d***N* or **D***N*, it has type **_Decimal***N*. If suffixed by **d***N***x** or **D***N***x**, it has type **_Decimal***N***x**.

[6] The quantum exponent of a floating constant of decimal floating type is the same as for the result value of the corresponding **strtod***N* or **strtod***N***x** function (X.12.2) for the same numeric string.

[7] **NOTE** For *N* = 32, 64, and 128, the suffixes **d***N* and **D***N* in this subclause for constants of type **_Decimal***N* are equivalent alternatives to the suffixes **df**, **dd**, **dl**, **DF**, **DD**, and **DL** in 6.4.4.2 for the same types.

X.6 Expressions

[1] This subclause expands the specification of expressions to also cover interchange and extended floating types.

[2] Operators involving operands of interchange or extended floating type are evaluated according to the semantics of IEC 60559, including production of decimal floating-point results with the preferred quantum exponent as specified in IEC 60559 (see 5.2.4.2.3).

[3] The *default argument promotions* (6.5.2.2) for functions whose type does not include a prototype are expanded so that arguments that have type **_Float16**, **_Float32**, or **_Float64** are promoted to **double**.

[4] For multiplicative operators (6.5.5), additive operators (6.5.6), relational operators (6.5.8), equality operators (6.5.9), and compound assignment operators (6.5.16.2), if either operand has decimal

floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

[5] For conditional operators (6.5.15), if the second or third operand has decimal floating type, the other of those operands shall not have standard floating type, binary floating type, complex type, or imaginary type.

[6] The equivalence of expressions noted in F.9.2 apply to expressions of binary floating types, as well as standard floating types.

X.7 Declarations

[1] This subclause expands the list of type specifiers (6.7.2) to also include:

> **_Float***N*, where *N* is 16, 32, 64, or ≥ 128 and a multiple of 32
> **_Float32x**
> **_Float64x**
> **_Float128x**
> **_Decimal***N*, where *N* is 96 or > 128 and a multiple of 32
> **_Decimal64x**
> **_Decimal128x**

[2] The type specifiers **_Float***N* (where *N* is 16, 32, 64, or ≥ 128 and a multiple of 32), **_Float32x**, **_Float64x**, **_Float128x**, **_Decimal***N* (where *N* is 96 or > 128 and a multiple of 32), **_Decimal64x**, and **_Decimal128x** shall not be used if the implementation does not support the corresponding types (see 6.10.8.3 and X.2).

[3] This subclause also expands the list under Constraints in 6.7.2 to also include:

— **_Float***N*, where *N* is 16, 32, 64 or ≥ 128 and a multiple of 32

— **_Float32x**

— **_Float64x**

— **_Float128x**

— **_Decimal***N*, where *N* is 96 or > 128 and a multiple of 32

— **_Decimal64x**

— **_Decimal128x**

— **_Float***N* **_Complex**, where *N* is 16, 32, 64, or ≥ 128 and a multiple of 32

— **_Float32x _Complex**

— **_Float64x _Complex**

— **_Float128x _Complex**

X.8 Identifiers in standard headers

[1] The identifiers added to library headers by this annex are defined or declared by their respective headers only if the macro **__STDC_WANT_IEC_60559_TYPES_EXT__** is defined (by the user) at the point in the code where the appropriate header is first included.

X.9 Complex arithmetic `<complex.h>`

[1] This subclause specifies complex functions for corresponding real types that are binary floating types.

[2] Each function synopsis in 7.3 specifies a family of functions including a principal function with one or more **double complex** parameters and a **double complex** or **double** return value. This subclause expands the synopsis to also include other functions, with the same name as the principal function but with **f***N* and **f***N***x** suffixes, which are corresponding functions whose parameters and return values have corresponding real types **_Float***N* and **_Float***N***x**.

[3] The following function prototypes are added to the synopses of the respective subclauses in 7.3. For each binary floating type that the implementation provides, **<complex.h>** shall declare the associated functions (see X.8). Conversely, for each such type that the implementation does not provide, **<complex.h>** shall not declare the associated functions.

7.3.5 Trigonometric functions

```
_FloatN complex cacosfN(_FloatN complex z);
_FloatNx complex cacosfNx(_FloatNx complex z);

_FloatN complex casinfN(_FloatN complex z);
_FloatNx complex casinfNx(_FloatNx complex z);

_FloatN complex catanfN(_FloatN complex z);
_FloatNx complex catanfNx(_FloatNx complex z);

_FloatN complex ccosfN(_FloatN complex z);
_FloatNx complex ccosfNx(_FloatNx complex z);

_FloatN complex csinfN(_FloatN complex z);
_FloatNx complex csinfNx(_FloatNx complex z);

_FloatN complex ctanfN(_FloatN complex z);
_FloatNx complex ctanfNx(_FloatNx complex z);
```

7.3.6 Hyperbolic functions

```
_FloatN complex cacoshfN(_FloatN complex z);
_FloatNx complex cacoshfNx(_FloatNx complex z);

_FloatN complex casinhfN(_FloatN complex z);
_FloatNx complex casinhfNx(_FloatNx complex z);

_FloatN complex catanhfN(_FloatN complex z);
_FloatNx complex catanhfNx(_FloatNx complex z);

_FloatN complex ccoshfN(_FloatN complex z);
_FloatNx complex ccoshfNx(_FloatNx complex z);

_FloatN complex csinhfN(_FloatN complex z);
_FloatNx complex csinhfNx(_FloatNx complex z);

_FloatN complex ctanhfN(_FloatN complex z);
_FloatNx complex ctanhfNx(_FloatNx complex z);
```

7.3.7 Exponential and logarithmic functions

```
_FloatN complex cexpfN(_FloatN complex z);
_FloatNx complex cexpfNx(_FloatNx complex z);

_FloatN complex clogfN(_FloatN complex z);
_FloatNx complex clogfNx(_FloatNx complex z);
```

7.3.8 Power and absolute value functions

```
_FloatN cabsfN(_FloatN complex z);
_FloatNx cabsfNx(_FloatNx complex z);

_FloatN complex cpowfN(_FloatN complex x,
  _FloatN complex y);
_FloatNx complex cpowfNx(_FloatNx complex x,
  _FloatNx complex y);

_FloatN complex csqrtfN(_FloatN complex z);
_FloatNx complex csqrtfNx(_FloatNx complex z);
```

7.3.9 Manipulation functions

```
_FloatN cargfN(_FloatN complex z);
_FloatNx cargfNx(_FloatNx complex z);

_FloatN cimagfN(_FloatN complex z);
_FloatNx cimagfNx(_FloatNx complex z);

_FloatN complex CMPLXFN(_FloatN x, _FloatN y);
_FloatNx complex CMPLXFNX(_FloatNx x, _FloatNx y);

_FloatN complex conjfN(_FloatN complex z);
_FloatNx complex conjfNx(_FloatNx complex z);

_FloatN complex cprojfN(_FloatN complex z);
_FloatNx complex cprojfNx(_FloatNx complex z);

_FloatN crealfN(_FloatN complex z);
_FloatNx crealfNx(_FloatNx complex z);
```

[4] For the functions listed in "future library directions" for **`<complex.h>`** (7.31.1), the possible suffixes are expanded to also include **`f`***N* and **`f`***N***`x`**.

X.10 Floating-point environment

[1] This subclause broadens the effects of the floating-point environment (7.6) to apply to types and formats specified in this annex.

[2] The same floating-point status flags are used by floating-point operations for all floating types, including those types introduced in this annex, and by conversions for IEC 60559 non-arithmetic interchange formats.

[3] Both the dynamic rounding direction mode accessed by **`fegetround`** and **`fesetround`** and the **`FENV_ROUND`** rounding control pragma apply to operations for binary floating types, as well as for standard floating types, and also to conversions for radix-2 non-arithmetic interchange formats.

Likewise, both the dynamic rounding direction mode accessed by **fe_dec_getround** and **fe_dec_setround** and the **FENV_DEC_ROUND** rounding control pragmas apply to operations for all the decimal floating types, including those decimal floating types introduced in this annex, and to conversions for radix-10 non-arithmetic interchange formats.

[4] In 7.6.2, the table of functions affected by constant rounding modes for standard floating types applies also for binary floating types. Each **<math.h>** function family listed in the table indicates the family of functions of all standard and binary floating types (for example, the **acos** family includes **acosf**, **acosl**, **acosf***N*, and **acosf***N***x** as well as **acos**). The **f***M***encf***N*, **strfromencf***N*, and **strtoencf***N* functions are also affected by these constant rounding modes.

[5] In 7.6.3, in the table of functions affected by constant rounding modes for decimal floating types, each **<math.h>** function family indicates the family of functions of all decimal floating types (for example, the **acos** family includes **acosd***N* and **acosd***N***x**). The **d***M***encbind***N*, **d***M***encdecd***N*, **strfromencbind***N*, **strfromencdecd***N*, **strtoencbind***N*, and **strtoencdecd***N* functions are also affected by these constant rounding modes.

X.11 Mathematics <math.h>

[1] This subclause specifies types, functions, and macros for interchange and extended floating types, generally corresponding to those specified in 7.12 and F.10.

[2] All classification macros (7.12.3) and comparison macros (7.12.17) naturally extend to handle interchange and extended floating types. For comparison macros, if neither of the sets of values of the argument formats is a subset of (or equivalent to) the other, the behavior is undefined.

[3] This subclause also specifies encoding conversion functions that are part of support for the non-arithmetic interchange formats in IEC 60559 (see X.2.2).

[4] Most function synopses in 7.12 specify a family of functions including a principal function with one or more **double** parameters, a **double** return value, or both. The synopses are expanded to also include functions with the same name as the principal function but with **f***N*, **f***N***x**, **d***N*, and **d***N***x** suffixes, which are corresponding functions whose parameters, return values, or both are of types **_Float***N*, **_Float***N***x**, **_Decimal***N*, and **_Decimal***N***x**, respectively.

[5] For each interchange or extended floating type that the implementation provides, **<math.h>** shall define the associated types and macros and declare the associated functions (see X.8). Conversely, for each such type that the implementation does not provide, **<math.h>** shall not define the associated types and macros or declare the associated functions unless explicitly specified otherwise.

[6] With the types

```
float_t
double_t
```

in 7.12 are included the type

```
long_double_t
```

and for each supported type **_Float***N*, the type

```
_FloatN_t
```

and for each supported type **_Decimal***N*, the type

```
_DecimalN_t
```


These are floating types, such that:

— each of the types has at least the range and precision of the corresponding real floating type;

— `long_double_t` has at least the range and precision of `double_t`;

— `_Float`*N*`_t` has at least the range and precision of `_Float`*M*`_t` if *N* > *M*;

— `_Decimal`*N*`_t` has at least the range and precision of `_Decimal`*M*`_t` if *N* > *M*.

If `FLT_RADIX` is 2 and `FLT_EVAL_METHOD` (X.3) is nonnegative, then each of the types corresponding to a standard or binary floating type is the type whose range and precision are specified by `FLT_EVAL_METHOD` to be used for evaluating operations and constants of that standard or binary floating type. If `DEC_EVAL_METHOD` (X.3) is nonnegative, then each of the types corresponding to a decimal floating type is the type whose range and precision are specified by `DEC_EVAL_METHOD` to be used for evaluating operations and constants of that decimal floating type.

[7] **EXAMPLE** If supported standard and binary floating types are

Type	IEC 60559 format
`_Float16`	binary16
`float`, `_Float32`	binary32
`double`, `_Float64`, `_Float32x`	binary64
`long double`, `_Float64x`	80-bit binary64-extended
`_Float128`	binary128

the following table gives the types with `_t` suffixes for various values for `FLT_EVAL_METHOD`.

	`_t` type as determined by `FLT_EVAL_METHOD` *m*							
`_t` type \ *m*	0	1	2	32	64	128	33	65
`_Float16_t`	`float`	`double`	`long double`	`_Float32`	`_Float64`	`_Float128`	`_Float32x`	`_Float64x`
`float_t`	`float`	`double`	`long double`	`float`	`_Float64`	`_Float128`	`_Float32x`	`_Float64x`
`_Float32_t`	`_Float32`	`double`	`long double`	`_Float32`	`_Float64`	`_Float128`	`_Float32x`	`_Float64x`
`double_t`	`double`	`double`	`long double`	`double`	`double`	`_Float128`	`double`	`_Float64x`
`_Float64_t`	`_Float64`	`_Float64`	`long double`	`_Float64`	`_Float64`	`_Float128`	`_Float64`	`_Float64x`
`long_double_t`	`long double`	`long double`	`long double`	`long double`	`long double`	`_Float128`	`long double`	`long double`
`_Float128_t`	`_Float128`	`_Float128`	`_Float128`	`_Float128`	`_Float128`	`_Float128`	`_Float128`	`_Float128`

X.11.1 Macros

[1] This subclause adds macros in 7.12 as follows.

[2] The macros

HUGE_VAL_F*N*
HUGE_VAL_D*N*
HUGE_VAL_F*N*X
HUGE_VAL_D*N*X

expand to constant expressions of types `_Float`*N*, `_Decimal`*N*, `_Float`*N*`x`, and `_Decimal`*N*`x`, respectively, representing positive infinity.

[3] The signaling NaN macros

SNANF*N*
SNAND*N*
SNANF*N***X**
SNAND*N***X**

expand to constant expressions of types **_Float***N*, **_Decimal***N*, **_Float***N***x**, and **_Decimal***N***x**, respectively, 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. [Note: This will have to change when WG14-approved N2476 is incorporated into draft C2X.]

[4] The macros

FP_FAST_FMAF*N*
FP_FAST_FMAD*N*
FP_FAST_FMAF*N***X**
FP_FAST_FMAD*N***X**

are, respectively, **_Float***N*, **_Decimal***N*, **_Float***N***x**, and **_Decimal***N***x** analogues of **FP_FAST_FMA**.

[5] The macros in the following lists are interchange and extended floating type analogues of **FP_FAST_FADD**, **FP_FAST_FADDL**, **FP_FAST_DADDL**, etc.

[6] For *M* < *N*, the macros

FP_FAST_F*M***ADDF***N*
FP_FAST_F*M***SUBF***N*
FP_FAST_F*M***MULF***N*
FP_FAST_F*M***DIVF***N*
FP_FAST_F*M***FMAF***N*
FP_FAST_F*M***SQRTF***N*
FP_FAST_D*M***ADDD***N*
FP_FAST_D*M***SUBD***N*
FP_FAST_D*M***MULD***N*
FP_FAST_D*M***DIVD***N*
FP_FAST_D*M***FMAD***N*
FP_FAST_D*M***SQRTD***N*

characterize the corresponding functions whose arguments are of an interchange floating type of width *N* and whose return type is an interchange floating type of width *M*.

[7] For $M \le N$, the macros

FP_FAST_F*M*ADDF*N*X
FP_FAST_F*M*SUBF*N*X
FP_FAST_F*M*MULF*N*X
FP_FAST_F*M*DIVF*N*X
FP_FAST_F*M*FMAF*N*X
FP_FAST_F*M*SQRTF*N*X
FP_FAST_D*M*ADDD*N*X
FP_FAST_D*M*SUBD*N*X
FP_FAST_D*M*MULD*N*X
FP_FAST_D*M*DIVD*N*X
FP_FAST_D*M*FMAD*N*X
FP_FAST_D*M*SQRTD*N*X

characterize the corresponding functions whose arguments are of an extended floating type that extends a format of width N and whose return type is an interchange floating type of width M.

[8] For $M < N$, the macros

FP_FAST_F*M*XADDF*N*
FP_FAST_F*M*XSUBF*N*
FP_FAST_F*M*XMULF*N*
FP_FAST_F*M*XDIVF*N*
FP_FAST_F*M*XFMAF*N*
FP_FAST_F*M*XSQRTF*N*
FP_FAST_D*M*XADDD*N*
FP_FAST_D*M*XSUBD*N*
FP_FAST_D*M*XMULD*N*
FP_FAST_D*M*XDIVD*N*
FP_FAST_D*M*XFMAD*N*
FP_FAST_D*M*XSQRTD*N*

characterize the corresponding functions whose arguments are of an interchange floating type of width N and whose return type is an extended floating type that extends a format of width M.

[9] For $M < N$, the macros

FP_FAST_F*M*XADDF*N*X
FP_FAST_F*M*XSUBF*N*X
FP_FAST_F*M*XMULF*N*X
FP_FAST_F*M*XDIVF*N*X
FP_FAST_F*M*XFMAF*N*X
FP_FAST_F*M*XSQRTF*N*X
FP_FAST_D*M*XADDD*N*X
FP_FAST_D*M*XSUBD*N*X
FP_FAST_D*M*XMULD*N*X
FP_FAST_D*M*XDIVD*N*X
FP_FAST_D*M*XFMAD*N*X
FP_FAST_D*M*XSQRTD*N*X

characterize the corresponding functions whose arguments are of an extended floating type that extends a format of width N and whose return type is an extended floating type that extends a format of width M.

X.11.2 Function prototypes

[1] This subclause adds the following function prototypes to the synopses of the respective subclauses in 7.12.

7.12.4 Trigonometric functions

```
_FloatN acosfN(_FloatN x);
_FloatNx acosfNx(_FloatNx x);
_DecimalN acosdN(_DecimalN x);
_DecimalNx acosdNx(_DecimalNx x);

_FloatN asinfN(_FloatN x);
_FloatNx asinfNx(_FloatNx x);
_DecimalN asindN(_DecimalN x);
_DecimalNx asindNx(_DecimalNx x);

_FloatN atanfN(_FloatN x);
_FloatNx atanfNx(_FloatNx x);
_DecimalN atandN(_DecimalN x);
_DecimalNx atandNx(_DecimalNx x);

_FloatN atan2fN(_FloatN y, _FloatN x);
_FloatNx atan2fNx(_FloatNx y, _FloatNx x);
_DecimalN atan2dN(_DecimalN y, _DecimalN x);
_DecimalNx atan2dNx(_DecimalNx y, _DecimalNx x);

_FloatN cosfN(_FloatN x);
_FloatNx cosfNx(_FloatNx x);
_DecimalN cosdN(_DecimalN x);
_DecimalNx cosdNx(_DecimalNx x);

_FloatN sinfN(_FloatN x);
_FloatNx sinfNx(_FloatNx x);
_DecimalN sindN(_DecimalN x);
_DecimalNx sindNx(_DecimalNx x);

_FloatN tanfN(_FloatN x);
_FloatNx tanfNx(_FloatNx x);
_DecimalN tandN(_DecimalN x);
_DecimalNx tandNx(_DecimalNx x);

_FloatN acospifN(_FloatN x);
_FloatNx acospifNx(_FloatNx x);
_DecimalN acospidN(_DecimalN x);
_DecimalNx acospidNx(_DecimalNx x);

_FloatN asinpifN(_FloatN x);
_FloatNx asinpifNx(_FloatNx x);
_DecimalN asinpidN(_DecimalN x);
_DecimalNx asinpidNx(_DecimalNx x);
```

```
_FloatN atanpifN(_FloatN x);
_FloatNx atanpifNx(_FloatNx x);
_DecimalN atanpidN(_DecimalN x);
_DecimalNx atanpidNx(_DecimalNx x);

_FloatN atan2pifN(_FloatN y, _FloatN x);
_FloatNx atan2pifNx(_FloatNx y, _FloatNx x);
_DecimalN atan2pidN(_DecimalN y, _DecimalN x);
_DecimalNx atan2pidNx(_DecimalNx y, _DecimalNx x);

_FloatN cospifN(_FloatN x);
_FloatNx cospifNx(_FloatNx x);
_DecimalN cospidN(_DecimalN x);
_DecimalNx cospidNx(_DecimalNx x);

_FloatN sinpifN(_FloatN x);
_FloatNx sinpifNx(_FloatNx x);
_DecimalN sinpidN(_DecimalN x);
_DecimalNx sinpidNx(_DecimalNx x);

_FloatN tanpifN(_FloatN x);
_FloatNx tanpifNx(_FloatNx x);
_DecimalN tanpidN(_DecimalN x);
_DecimalNx tanpidNx(_DecimalNx x);
```

7.12.5 Hyperbolic functions

```
_FloatN acoshfN(_FloatN x);
_FloatNx acoshfNx(_FloatNx x);
_DecimalN acoshdN(_DecimalN x);
_DecimalNx acoshdNx(_DecimalNx x);

_FloatN asinhfN(_FloatN x);
_FloatNx asinhfNx(_FloatNx x);
_DecimalN asinhdN(_DecimalN x);
_DecimalNx asinhdNx(_DecimalNx x);

_FloatN atanhfN(_FloatN x);
_FloatNx atanhfNx(_FloatNx x);
_DecimalN atanhdN(_DecimalN x);
_DecimalNx atanhdNx(_DecimalNx x);

_FloatN coshfN(_FloatN x);
_FloatNx coshfNx(_FloatNx x);
_DecimalN coshdN(_DecimalN x);
_DecimalNx coshdNx(_DecimalNx x);

_FloatN sinhfN(_FloatN x);
_FloatNx sinhfNx(_FloatNx x);
_DecimalN sinhdN(_DecimalN x);
_DecimalNx sinhdNx(_DecimalNx x);
```

```
_FloatN tanhfN(_FloatN x);
_FloatNx tanhfNx(_FloatNx x);
_DecimalN tanhdN(_DecimalN x);
_DecimalNx tanhdNx(_DecimalNx x);
```

7.12.6 Exponential and logarithmic functions

```
_FloatN expfN(_FloatN x);
_FloatNx expfNx(_FloatNx x);
_DecimalN expdN(_DecimalN x);
_DecimalNx expdNx(_DecimalNx x);

_FloatN exp10fN(_FloatN x);
_FloatNx exp10fNx(_FloatNx x);
_DecimalN exp10dN(_DecimalN x);
_DecimalNx exp10dNx(_DecimalNx x);

_FloatN exp10m1fN(_FloatN x);
_FloatNx exp10m1fNx(_FloatNx x);
_DecimalN exp10m1dN(_DecimalN x);
_DecimalNx exp10m1dNx(_DecimalNx x);

_FloatN exp2fN(_FloatN x);
_FloatNx exp2fNx(_FloatNx x);
_DecimalN exp2dN(_DecimalN x);
_DecimalNx exp2dNx(_DecimalNx x);

_FloatN exp2m1fN(_FloatN x);
_FloatNx exp2m1fNx(_FloatNx x);
_DecimalN exp2m1dN(_DecimalN x);
_DecimalNx exp2m1dNx(_DecimalNx x);

_FloatN expm1fN(_FloatN x);
_FloatNx expm1fNx(_FloatNx x);
_DecimalN expm1dN(_DecimalN x);
_DecimalNx expm1dNx(_DecimalNx x);

_FloatN frexpfN(_FloatN value, int *exp);
_FloatNx frexpfNx(_FloatNx value, int *exp);
_DecimalN frexpdN(_DecimalN value, int *exp);
_DecimalNx frexpdNx(_DecimalNx value, int *exp);

int ilogbfN(_FloatN x);
int ilogbfNx(_FloatNx x);
int ilogbdN(_DecimalN x);
int ilogbdNx(_DecimalNx x);

_FloatN ldexpfN(_FloatN value, int exp);
_FloatNx ldexpfNx(_FloatNx value, int exp);
_DecimalN ldexpdN(_DecimalN value, int exp);
_DecimalNx ldexpdNx(_DecimalNx value, int exp);
```

```
long int llogbfN(_FloatN x);
long int llogbfNx(_FloatNx x);
long int llogbdN(_DecimalN x);
long int llogbdNx(_DecimalNx x);

_FloatN logfN(_FloatN x);
_FloatNx logfNx(_FloatNx x);
_DecimalN logdN(_DecimalN x);
_DecimalNx logdNx(_DecimalNx x);

_FloatN log10fN(_FloatN x);
_FloatNx log10fNx(_FloatNx x);
_DecimalN log10dN(_DecimalN x);
_DecimalNx log10dNx(_DecimalNx x);

_FloatN log10p1fN(_FloatN x);
_FloatNx log10p1fNx(_FloatNx x);
_DecimalN log10p1dN(_DecimalN x);
_DecimalNx log10p1dNx(_DecimalNx x);

_FloatN log1pfN(_FloatN x);
_FloatNx log1pfNx(_FloatNx x);
_FloatN logp1fN(_FloatN x);
_FloatNx logp1fNx(_FloatNx x);
_DecimalN log1pdN(_DecimalN x);
_DecimalNx log1pdNx(_DecimalNx x);
_DecimalN logp1dN(_DecimalN x);
_DecimalNx logp1dNx(_DecimalNx x);

_FloatN log2fN(_FloatN x);
_FloatNx log2fNx(_FloatNx x);
_DecimalN log2dN(_DecimalN x);
_DecimalNx log2dNx(_DecimalNx x);

_FloatN log2p1fN(_FloatN x);
_FloatNx log2p1fNx(_FloatNx x);
_DecimalN log2p1dN(_DecimalN x);
_DecimalNx log2p1dNx(_DecimalNx x);

_FloatN logbfN(_FloatN x);
_FloatNx logbfNx(_FloatNx x);
_DecimalN logbdN(_DecimalN x);
_DecimalNx logbdNx(_DecimalNx x);

_FloatN modffN(_FloatN x, _FloatN *iptr);
_FloatNx modffNx(_FloatNx x, _FloatNx *iptr);
_DecimalN modfdN(_DecimalN x, _DecimalN *iptr);
_DecimalNx modfdNx(_DecimalNx x, _DecimalNx *iptr);

_FloatN scalbnfN(_FloatN value, int exp);
_FloatNx scalbnfNx(_FloatNx value, int exp);
_DecimalN scalbndN(_DecimalN value, int exp);
_DecimalNx scalbndNx(_DecimalNx value, int exp);
```

```
_FloatN scalblnfN(_FloatN value, long int exp);
_FloatNx scalblnfNx(_FloatNx value, long int exp);
_DecimalN scalblndN(_DecimalN value, long int exp);
_DecimalNx scalblndNx(_DecimalNx value, long int exp);
```

7.12.7 Power and absolute-value functions

```
_FloatN cbrtfN(_FloatN x);
_FloatNx cbrtfNx(_FloatNx x);
_DecimalN cbrtdN(_DecimalN x);
_DecimalNx cbrtdNx(_DecimalNx x);

_FloatN compoundnfN(_FloatN x, intmax_t n);
_FloatNx compoundnfNx(_FloatNx x, intmax_t n);
_DecimalN compoundndN(_DecimalN x, intmax_t n);
_DecimalNx compoundndNx(_DecimalNx x, intmax_t n);

_FloatN fabsfN(_FloatN x);
_FloatNx fabsfNx(_FloatNx x);
_DecimalN fabsdN(_DecimalN x);
_DecimalNx fabsdNx(_DecimalNx x);

_FloatN hypotfN(_FloatN x, _FloatN y);
_FloatNx hypotfNx(_FloatNx x, _FloatNx y);
_DecimalN hypotdN(_DecimalN x, _DecimalN y);
_DecimalNx hypotdNx(_DecimalNx x, _DecimalNx y);

_FloatN powfN(_FloatN x, _FloatN y);
_FloatNx powfNx(_FloatNx x, _FloatNx y);
_DecimalN powdN(_DecimalN x, _DecimalN y);
_DecimalNx powdNx(_DecimalNx x, _DecimalNx y);

_FloatN pownfN(_FloatN x, intmax_t n);
_FloatNx pownfNx(_FloatNx x, intmax_t n);
_DecimalN powndN(_DecimalN x, intmax_t n);
_DecimalNx powndNx(_DecimalNx x, intmax_t n);

_FloatN powrfN(_FloatN x, _FloatN y);
_FloatNx powrfNx(_FloatNx x, _FloatNx y);
_DecimalN powrdN(_DecimalN x, _DecimalN y);
_DecimalNx powrdNx(_DecimalNx x, _DecimalNx y);

_FloatN rootnfN(_FloatN x, intmax_t n);
_FloatNx rootnfNx(_FloatNx x, intmax_t n);
_DecimalN rootndN(_DecimalN x, intmax_t n);
_DecimalNx rootndNx(_DecimalNx x, intmax_t n);

_FloatN rsqrtfN(_FloatN x);
_FloatNx rsqrtfNx(_FloatNx x);
_DecimalN rsqrtdN(_DecimalN x);
_DecimalNx rsqrtdNx(_DecimalNx x);
```



```
_FloatN sqrtfN(_FloatN x);
_FloatNx sqrtfNx(_FloatNx x);
_DecimalN sqrtdN(_DecimalN x);
_DecimalNx sqrtdNx(_DecimalNx x);
```

7.12.8 Error and gamma functions

```
_FloatN erffN(_FloatN x);
_FloatNx erffNx(_FloatNx x);
_DecimalN erfdN(_DecimalN x);
_DecimalNx erfdNx(_DecimalNx x);

_FloatN erfcfN(_FloatN x);
_FloatNx erfcfNx(_FloatNx x);
_DecimalN erfcdN(_DecimalN x);
_DecimalNx erfcdNx(_DecimalNx x);

_FloatN lgammafN(_FloatN x);
_FloatNx lgammafNx(_FloatNx x);
_DecimalN lgammadN(_DecimalN x);
_DecimalNx lgammadNx(_DecimalNx x);

_FloatN tgammafN(_FloatN x);
_FloatNx tgammafNx(_FloatNx x);
_DecimalN tgammadN(_DecimalN x);
_DecimalNx tgammadNx(_DecimalNx x);
```

7.12.9 Nearest integer functions

```
_FloatN ceilfN(_FloatN x);
_FloatNx ceilfNx(_FloatNx x);
_DecimalN ceildN(_DecimalN x);
_DecimalNx ceildNx(_DecimalNx x);

_FloatN floorfN(_FloatN x);
_FloatNx floorfNx(_FloatNx x);
_DecimalN floordN(_DecimalN x);
_DecimalNx floordNx(_DecimalNx x);

_FloatN nearbyintfN(_FloatN x);
_FloatNx nearbyintfNx(_FloatNx x);
_DecimalN nearbyintdN(_DecimalN x);
_DecimalNx nearbyintdNx(_DecimalNx x);

_FloatN rintfN(_FloatN x);
_FloatNx rintfNx(_FloatNx x);
_DecimalN rintdN(_DecimalN x);
_DecimalNx rintdNx(_DecimalNx x);

long int lrintfN(_FloatN x);
long int lrintfNx(_FloatNx x);
long int lrintdN(_DecimalN x);
long int lrintdNx(_DecimalNx x);
```

```
long long int llrintfN(_FloatN x);
long long int llrintfNx(_FloatNx x);
long long int llrintdN(_DecimalN x);
long long int llrintdNx(_DecimalNx x);

_FloatN roundfN(_FloatN x);
_FloatNx roundfNx(_FloatNx x);
_DecimalN rounddN(_DecimalN x);
_DecimalNx rounddNx(_DecimalNx x);

long int lroundfN(_FloatN x);
long int lroundfNx(_FloatNx x);
long int lrounddN(_DecimalN x);
long int lrounddNx(_DecimalNx x);

long long int llroundfN(_FloatN x);
long long int llroundfNx(_FloatNx x);
long long int llrounddN(_DecimalN x);
long long int llrounddNx(_DecimalNx x);

_FloatN roundevenfN(_FloatN x);
_FloatNx roundevenfNx(_FloatNx x);
_DecimalN roundevendN(_DecimalN x);
_DecimalNx roundevendNx(_DecimalNx x);

_FloatN truncfN(_FloatN x);
_FloatNx truncfNx(_FloatNx x);
_DecimalN truncdN(_DecimalN x);
_DecimalNx truncdNx(_DecimalNx x);

intmax_t fromfpfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpfNx(_FloatNx x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpdNx(_DecimalNx x, int round,
  unsigned int width);
uintmax_t ufromfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfNx(_FloatNx x, int round,
  unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round,
  unsigned int width);
uintmax_t ufromfpdNx(_DecimalNx x, int round,
  unsigned int width);

intmax_t fromfpxfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpxfNx(_FloatNx x, int round, unsigned int width);
intmax_t fromfpxdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpxdNx(_DecimalNx x, int round,
  unsigned int width);
uintmax_t ufromfpxfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpxfNx(_FloatNx x, int round,
  unsigned int width);
uintmax_t ufromfpxdN(_DecimalN x, int round,
  unsigned int width);
uintmax_t ufromfpxdNx(_DecimalNx x, int round,
  unsigned int width);
```

7.12.10 Remainder functions

```
_FloatN fmodfN(_FloatN x,_FloatN y);
_FloatNx fmodfNx(_FloatNx x,_FloatNx y);
_DecimalN fmoddN(_DecimalN x,_DecimalN y);
_DecimalNx fmoddNx(_DecimalNx x,_DecimalNx y);

_FloatN remainderfN(_FloatN x,_FloatN y);
_FloatNx remainderfNx(_FloatNx x,_FloatNx y);
_DecimalN remainderdN(_DecimalN x,_DecimalN y);
_DecimalNx remainderdNx(_DecimalNx x,_DecimalNx y);

_FloatN remquofN(_FloatN x,_FloatN y, int *quo);
_FloatNx remquofNx(_FloatNx x,_FloatNx y, int *quo);
```

7.12.11 Manipulation functions

```
_FloatN copysignfN(_FloatN x,_FloatN y);
_FloatNx copysignfNx(_FloatNx x,_FloatNx y);
_DecimalN copysigndN(_DecimalN x,_DecimalN y);
_DecimalNx copysigndNx(_DecimalNx x,_DecimalNx y);

_FloatN nanfN(const char *tagp);
_FloatNx nanfNx(const char *tagp);
_DecimalN nandN(const char *tagp);
_DecimalNx nandNx(const char *tagp);

_FloatN nextafterfN(_FloatN x,_FloatN y);
_FloatNx nextafterfNx(_FloatNx x,_FloatNx y);
_DecimalN nextafterdN(_DecimalN x,_DecimalN y);
_DecimalNx nextafterdNx(_DecimalNx x,_DecimalNx y);

_FloatN nextupfN(_FloatN x);
_FloatNx nextupfNx(_FloatNx x);
_DecimalN nextupdN(_DecimalN x);
_DecimalNx nextupdNx(_DecimalNx x);

_FloatN nextdownfN(_FloatN x);
_FloatNx nextdownfNx(_FloatNx x);
_DecimalN nextdowndN(_DecimalN x);
_DecimalNx nextdowndNx(_DecimalNx x);

int canonicalizefN(_FloatN * cx, const _FloatN * x);
int canonicalizefNx(_FloatNx * cx, const _FloatNx * x);
int canonicalizedN(_DecimalN * cx, const _DecimalN * x);
int canonicalizedNx(_DecimalNx * cx, const _DecimalNx * x);
```

7.12.12 Maximum, minimum, and positive difference functions

```
_FloatN fdimfN(_FloatN x,_FloatN y);
_FloatNx fdimfNx(_FloatNx x,_FloatNx y);
_DecimalN fdimdN(_DecimalN x,_DecimalN y);
_DecimalNx fdimdNx(_DecimalNx x,_DecimalNx y);
```

```
_FloatN fmaxfN(_FloatN x, _FloatN y);
_FloatNx fmaxfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaxdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaxdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminfN(_FloatN x, _FloatN y);
_FloatNx fminfNx(_FloatNx x, _FloatNx y);
_DecimalN fmindN(_DecimalN x, _DecimalN y);
_DecimalNx fmindNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaxmagfN(_FloatN x, _FloatN y);
_FloatNx fmaxmagfNx(_FloatNx x, _FloatNx y);
_DecimalN fmaxmagdN(_DecimalN x, _DecimalN y);
_DecimalNx fmaxmagdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminmagfN(_FloatN x, _FloatN y);
_FloatNx fminmagfNx(_FloatNx x, _FloatNx y);
_DecimalN fminmagdN(_DecimalN x, _DecimalN y);
_DecimalNx fminmagdNx(_DecimalNx x, _DecimalNx y);
```

7.12.13 Floating multiply-add

```
_FloatN fmafN(_FloatN x, _FloatN y, _FloatN z);
_FloatNx fmafNx(_FloatNx x, _FloatNx y, _FloatNx z);
_DecimalN fmadN(_DecimalN x, _DecimalN y, _DecimalN z);
_DecimalNx fmadNx(_DecimalNx x, _DecimalNx y, _DecimalNx z);
```

7.12.14 Functions that round result to narrower type

```
_FloatM fMaddfN(_FloatN x, _FloatN y);   // M < N
_FloatM fMaddfNx(_FloatNx x, _FloatNx y);   // M <= N
_FloatMx fMxaddfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxaddfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMadddN(_DecimalN x, _DecimalN y);   // M < N
_DecimalM dMadddNx(_DecimalNx x, _DecimalNx y);   // M <= N
_DecimalMx dMxadddN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxadddNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMsubfN(_FloatN x, _FloatN y);   // M < N
_FloatM fMsubfNx(_FloatNx x, _FloatNx y);   // M <= N
_FloatMx fMxsubfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxsubfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMsubdN(_DecimalN x, _DecimalN y);   // M < N
_DecimalM dMsubdNx(_DecimalNx x, _DecimalNx y);   // M <= N
_DecimalMx dMxsubdN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxsubdNx(_DecimalNx x, _DecimalNx y); // M < N
```

```
_FloatM fMmulfN(_FloatN x, _FloatN y);   // M < N
_FloatM fMmulfNx(_FloatNx x, _FloatNx y);   // M <= N
_FloatMx fMxmulfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxmulfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMmuldN(_DecimalN x, _DecimalN y);   // M < N
_DecimalM dMmuldNx(_DecimalNx x, _DecimalNx y);   // M <= N
_DecimalMx dMxmuldN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxmuldNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMdivfN(_FloatN x, _FloatN y);   // M < N
_FloatM fMdivfNx(_FloatNx x, _FloatNx y);   // M <= N
_FloatMx fMxdivfN(_FloatN x, _FloatN y); // M < N
_FloatMx fMxdivfNx(_FloatNx x, _FloatNx y); // M < N
_DecimalM dMdivdN(_DecimalN x, _DecimalN y);   // M < N
_DecimalM dMdivdNx(_DecimalNx x, _DecimalNx y);   // M <= N
_DecimalMx dMxdivdN(_DecimalN x, _DecimalN y); // M < N
_DecimalMx dMxdivdNx(_DecimalNx x, _DecimalNx y); // M < N

_FloatM fMfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMfmafNx(_FloatNx x, _FloatNx y,
  _FloatNx z);   // M <= N
_FloatMx fMxfmafN(_FloatN x, _FloatN y, _FloatN z);  // M < N
_FloatMx fMxfmafNx(_FloatNx x, _FloatNx y,
  _FloatNx z);   // M < N
_DecimalM dMfmadN(_DecimalN x, _DecimalN y,
  _DecimalN z);  // M < N
_DecimalM dMfmadNx(_DecimalNx x, _DecimalNx y,
  _DecimalNx z); // M <= N
_DecimalMx dMxfmadN(_DecimalN x, _DecimalN y,
  _DecimalN z);  // M < N
_DecimalMx dMxfmadNx(_DecimalNx x, _DecimalNx y,
  _DecimalNx z); // M < N

_FloatM fMsqrtfN(_FloatN x); // M < N
_FloatM fMsqrtfNx(_FloatNx x);  // M <= N
_FloatMx fMxsqrtfN(_FloatN x);  // M < N
_FloatMx fMxsqrtfNx(_FloatNx x);   // M < N
_DecimalM dMsqrtdN(_DecimalN x);   // M < N
_DecimalM dMsqrtdNx(_DecimalNx x); // M <= N
_DecimalMx dMxsqrtdN(_DecimalN x); // M < N
_DecimalMx dMxsqrtdNx(_DecimalNx x);  // M < N
```

7.12.15 Quantum and quantum exponent functions

```
_DecimalN quantizedN(_DecimalN x,_DecimalN y);
_DecimalNx quantizedNx(_DecimalNx x,_DecimalNx y);

_Bool samequantumdN(_DecimalN x,_DecimalN y);
_Bool samequantumdNx(_DecimalNx x,_DecimalNx y);

_DecimalN quantumdN(_DecimalN x);
_DecimalNx quantumdNx(_DecimalNx x);
```



```
long long int llquantexpdN(_DecimalN x);
long long int llquantexpdNx(_DecimalNx x);
```

7.12.16 Decimal re-encoding functions

```
void encodedecdN(unsigned char * restrict encptr,
  const _DecimalN * restrict xptr);

void decodedecdN(_DecimalN * restrict xptr,
  const unsigned char * restrict encptr);

void encodebindN(unsigned char * restrict encptr,
  const _DecimalN * restrict xptr);

void decodebindN(_DecimalN * restrict xptr,
  const unsigned char * restrict encptr);
```

F.10.12 Total order functions

```
int totalorderfN(const _FloatN *x, const _FloatN *y);
int totalorderfNx(const _FloatNx *x, const _FloatNx *y);
int totalorderdN(const _DecimalN *x, const _DecimalN *y);
int totalorderdNx(const _DecimalNx *x, const _DecimalNx *y);

int totalordermagfN(const _FloatN *x, const _FloatN *y);
int totalordermagfNx(const _FloatNx *x, const _FloatNx *y);
int totalordermagdN(const _DecimalN *x, const _DecimalN *y);
int totalordermagdNx(const _DecimalNx *x, const _DecimalNx *y);
```

F.10.13 Payload functions

```
_FloatN getpayloadfN(const _FloatN *x);
_FloatNx getpayloadfNx(const _FloatNx *x);
_DecimalN getpayloaddN(const _DecimalN *x);
_DecimalNx getpayloaddNx(const _DecimalNx *x);

int setpayloadfN(_FloatN *res, _FloatN pl);
int setpayloadfNx(_FloatNx *res, _FloatNx pl);
int setpayloaddN(_DecimalN *res, _DecimalN pl);
int setpayloaddNx(_DecimalNx *res, _DecimalNx pl);

int setpayloadsigfN(_FloatN *res, _FloatN pl);
int setpayloadsigfNx(_FloatNx *res, _FloatNx pl);
int setpayloadsigdN(_DecimalN *res, _DecimalN pl);
int setpayloadsigdNx(_DecimalNx *res, _DecimalNx pl);
```

[2] The specification of the **frexp** functions (7.12.6.7) applies to the functions for binary floating types like those for standard floating types: the exponent is an integral power of 2 and, when applicable, **value** equals **x** × $2^{\texttt{*exp}}$.

[3] The specification of the **ldexp** functions (7.12.6.9) applies to the functions for binary floating types like those for standard floating types: they return **x** × $2^{\texttt{exp}}$.

[4] The specification of the **logb** functions (7.12.6.17) applies to binary floating types, with b = 2.

[5] The specification of the **scalbn** and **scalbln** functions (7.12.6.19) applies to binary floating types, with $b = 2$.

X.11.3 Encoding conversion functions

[1] This subclause introduces **<math.h>** functions that, together with the numerical conversion functions for encodings in X.12, support the non-arithmetic interchange formats specified by IEC 60559. Support for these formats is an optional feature of this annex. Implementations that do not support non-arithmetic interchange formats need not declare the functions in this subclause.

[2] Non-arithmetic interchange formats are not associated with floating types. Arrays of element type **unsigned char** are used as parameters for conversion functions, to represent encodings in interchange formats that might be non-arithmetic formats.

X.11.3.1 Encode and decode functions

[1] This subclause specifies functions to map representations in binary floating types to and from encodings in **unsigned char** arrays.

X.11.3.1.1 The encodef*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
void encodefN(unsigned char encptr[restrict static N/8],
   const _FloatN * restrict xptr);
```

Description

[2] The **encodef***N* functions convert ***xptr** into an IEC 60559 binary*N* encoding and store the resulting encoding as an *N*/8 element array, with 8 bits per array element, in the object pointed to by **encptr**. The order of bytes in the array is implementation-defined. These functions preserve the value of ***xptr** and raise no floating-point exceptions. If ***xptr** is non-canonical, these functions may or may not produce a canonical encoding.

Returns

[3] The **encodef***N* functions return no value.

X.11.3.1.2 The decodef*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
void decodefN (_FloatN * restrict xptr,
   const unsigned char encptr[restrict static N/8]);
```

Description

[2] The **decodef***N* functions interpret the *N*/8 element array pointed to by **encptr** as an IEC 60559 binary*N* encoding, with 8 bits per array element. The order of bytes in the array is implementation-defined. These functions convert the given encoding into a representation in the type **_Float***N*, and store the result in the object pointed to by **xptr**. These functions preserve the encoded value and raise no floating-point exceptions. If the encoding is non-canonical, these functions may or may not produce a canonical representation.

Returns

[3] The **decodef**_N_ functions return no value.

[4] See **EXAMPLE** in X.11.3.2.1.

X.11.3.2 Encoding-to-encoding conversion functions

[1] An implementation shall declare an **f**_M_**encf**_N_ function for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format, $M \neq N$. An implementation shall provide both **d**_M_**encdecd**_N_ and **d**_M_**encbind**_N_ functions for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format, $M \neq N$.

X.11.3.2.1 The f*M*encf*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
void fMencfN(unsigned char encMptr[restrict static M/8],
   const unsigned char encNptr[restrict static N/8]);
```

Description

[2] These functions convert between IEC 60559 binary interchange formats. These functions interpret the $N/8$ element array pointed to by **enc**_N_**ptr** as an encoding of width N bits. They convert the encoding to an encoding of width M bits and store the resulting encoding as an $M/8$ element array in the object pointed to by **enc**_M_**ptr**. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

[4] **EXAMPLE** If the IEC 60559 binary16 format is supported as a non-arithmetic format, data in binary16 format can be converted to type **float** as follows:

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
unsigned char b16[2];    // for input binary16 datum
float f;                 // for result
unsigned char b32[4];
_Float32 f32;
// store input binary16 datum in array b16
...
f32encf16(b32, b16);
decodef32(&f32, b32);
f = f32;
...
```

X.11.3.2.2 The d*M*encdecd*N* and d*M*encbind*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>
void dMencdecdN(unsigned char encMptr[restrict static M/8],
   const unsigned char encNptr[restrict static N/8]);
void dMencbindN(unsigned char encMptr[restrict static M/8],
   const unsigned char encNptr[restrict static N/8]);
```

Description

[2] These functions convert between IEC 60559 decimal interchange formats that use the same encoding scheme. The **d*M*encdecd*N*** functions convert between formats using the encoding scheme based on decimal encoding of the significand. The **d*M*encbind*N*** functions convert between formats using the encoding scheme based on binary encoding of the significand. These functions interpret the *N*/8 element array pointed to by **enc*N*ptr** as an encoding of width *N* bits. They convert the encoding to an encoding of width M bits and store the resulting encoding as an *M*/8 element array in the object pointed to by **enc*M*ptr**. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

X.12 Numeric conversion functions in **<stdlib.h>**

[1] This clause expands the specification of numeric conversion functions in **<stdlib.h>** (7.22.1) to also include conversions of strings from and to interchange and extended floating types. The conversions from floating are provided by functions analogous to the **strfromd** function. The conversions to floating are provided by functions analogous to the **strtod** function.

[2] This clause also specifies functions to convert strings from and to IEC 60559 interchange format encodings.

[3] For each interchange or extended floating type that the implementation provides, **<stdlib.h>** shall declare the associated functions specified below in X.12.1 and X.12.2 (see X.8). Conversely, for each such type that the implementation does not provide, **<stdlib.h>** shall not declare the associated functions.

[4] For each IEC 60559 arithmetic or non-arithmetic format that the implementation supports, **<stdlib.h>** shall declare the associated functions specified below in X.12.3 and X.12.4 (see X.8). Conversely, for each such format that the implementation does not provide, **<stdlib.h>** shall not declare the associated functions.

X.12.1 String from floating

[1] This subclause expands 7.22.1.3 and 7.22.1.4 to also include functions for the interchange and extended floating types. It adds to the synopsis in 7.22.1.3 the prototypes

```
int strfromfN(char * restrict s, size_t n,
   const char * restrict format, _FloatN fp);
int strfromfNx(char * restrict s, size_t n,
   const char * restrict format, _FloatNx fp);
```

It encompasses the prototypes in 7.22.1.4 by replacing them with

```
int strfromdN(char * restrict s, size_t n,
   const char * restrict format, _DecimalN fp);
int strfromdNx(char * restrict s, size_t n,
   const char * restrict format, _DecimalNx fp);
```

[2] The descriptions and returns for the added functions are analogous to the ones in 7.22.1.3 and 7.22.1.4.

X.12.2 String to floating

[1] This subclause expands 7.22.1.5 and 7.22.1.6 to also include functions for the interchange and extended floating types. It adds to the synopsis in 7.22.1.5 the prototypes

```
_FloatN strtofN(const char * restrict nptr,
   char ** restrict endptr);
_FloatNx strtofNx(const char * restrict nptr,
   char ** restrict endptr);
```

It encompasses the prototypes in 7.22.1.6 by replacing them with

```
_DecimalN strtodN(const char * restrict nptr,
   char ** restrict endptr);
_DecimalNx strtodNx(const char * restrict nptr,
   char ** restrict endptr);
```

[2] The descriptions and returns for the added functions are analogous to the ones in 7.22.1.5 and 7.22.1.6.

[3] For implementations that support both binary and decimal floating types and a (binary or decimal) non-arithmetic interchange format, the **strtod***N* and **strtod***N***x** functions (and hence the **strtoencdecd***N* and **strtoencbind***N* functions in X.12.4.2) shall accept subject sequences that have the form of hexadecimal floating numbers and otherwise meet the requirements of subject sequences (7.22.1.6). Then the decimal results shall be correctly rounded if the subject sequence has at most M significant hexadecimal digits, where $M \geq \lceil (P-1)/4 \rceil + 1$ is implementation defined, and P is the maximum precision of the supported binary floating types and binary non-arithmetic formats. If all subject sequences of hexadecimal form are correctly rounded, M may be regarded as infinite. If the subject sequence has more than M significant hexadecimal digits, the implementation may first round to M significant hexadecimal digits according to the applicable rounding direction mode, signaling exceptions as though converting from a wider format, then correctly round the result of the shortened hexadecimal input to the result type.

[4] **EXAMPLE** If the IEC 60559 binary128 format is supported as a non-arithmetic format, data in binary128 format can be converted to type **_Decimal128** as follows:

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
#define MAXSIZE 41        // > intermediate hex string length
unsigned char b128[16];   // for input binary128 datum
_Decimal128 d128;         // for result
char s[MAXSIZE];
// store input binary128 datum in array b128
...
strfromencf128(s, MAXSIZE, "%a", b128);
d128 = strtod128(s, NULL);
...
```

Use of "**%a**" for formatting assures an exact conversion of the value in binary format to character sequence. The value of that character sequence will be correctly rounded to **_Decimal128**, as specified above in this subclause. The array **s** for the output of **strfromencf128** need have no greater size than 41, which is the maximum length of strings of the form

[-]**0x***h***.***h*…*h***p**±*d*, where there are up to 29 hexadecimal digits *h* and *d* has 5 digits

plus 1 for the null character.

X.12.3 String from encoding

[1] An implementation shall declare the **strfromencf***N* function for each *N* equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the **strfromencdec***N* and **strfromencbind***N* functions for each N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

X.12.3.1 The strfromencf*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
int strfromencfN(char * restrict s, size_t n,
   const char * restrict format,
   const unsigned char encptr[restrict static N/8]);
```

Description

[2] The **strfromencf***N* functions are similar to the **strfromf***N* functions, except the input is the value of the *N*/8 element array pointed to by **encptr**, interpreted as an IEC 60559 binary*N* encoding. The order of bytes in the arrays is implementation-defined.

Returns

[3] The **strfromencf***N* functions return the same values as corresponding **strfromf***N* functions.

X.12.3.2 The strfromencdecd*N* and strfromencbind*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
int strfromencdecdN(char * restrict s, size_t n,
   const char * restrict format,
   const unsigned char encptr[restrict static N/8]);
int strfromencbindNx(char * restrict s, size_t n,
   const char * restrict format,
   const unsigned char encptr[restrict static N/8]);
```

Description

[2] The **strfromencdecd***N* functions are similar to the **strfromd***N* functions except the input is the value of the *N*/8 element array pointed to by **encptr**, interpreted as an IEC 60559 decimal*N* encoding in the coding scheme based on decimal encoding of the significand. The **strfromencbind***N* functions are similar to the **strfromd***N* functions except the input is the value of the *N*/8 element array pointed to by **encptr**, interpreted as an IEC 60559 decimal*N* encoding in the coding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

Returns

[3] The **`strfromencdecd`***N* and **`strfromencbind`***N* functions return the same values as corresponding **`strfromd`***N* functions.

X.12.4 String to encoding

[1] An implementation shall declare the **`strtoencf`***N* function for each *N* equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the **`strtoencdecd`***N* and **`strtoencbind`***N* functions for each N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.

X.12.4.1 The `strtoencf`*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
void strtoencfN(unsigned char encptr[restrict static N/8],
   const char * restrict nptr, char ** restrict endptr);
```

Description

[2] The **`strtoencf`***N* functions are similar to the **`strtof`***N* functions, except they store an IEC 60559 encoding of the result as an *N*/8 element array in the object pointed to by **`encptr`**. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

X.12.4.2 The `strtoencdecd`*N* and `strtoencbind`*N* functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
void strtoencdecdN(unsigned char encptr[restrict static N/8],
   const char * restrict nptr, char ** restrict endptr);
void strtoencbindN(unsigned char encptr[restrict static N/8],
   const char * restrict nptr, char ** restrict endptr);
```

Description

[2] The **`strtoencdecd`***N* and **`strtoencbind`***N* functions are similar to the **`strtod`***N* functions, except they store an IEC 60559 encoding of the result as an *N*/8 element array in the object pointed to by **`encptr`**. The **`strtoencdecd`***N* functions produce an encoding in the encoding scheme based on decimal encoding of the significand. The **`strtoencbind`***N* functions produce an encoding in the encoding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

X.13 Type-generic macros **<tgmath.h>**

[1] This clause enhances the specification of type-generic macros in **<tgmath.h>** (7.25) to apply to interchange and extended floating types, as well as standard floating types.

[2] If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is a standard floating type or a binary floating type and another argument is of decimal floating type, the behavior is undefined.

[3] The treatment of arguments of integer type in 7.25 is expanded to cases where another argument has extended type. Arguments of integer type are regarded as having type:

_Decimal64x	if any argument has a decimal extended type; otherwise
_Float32x	if any argument has a binary extended type; otherwise
_Decimal64	if any argument has decimal type; otherwise
double	

[4] Use of the macro **carg**, **cimag**, **conj**, **cproj**, or **creal** with any argument of standard floating type, binary floating type, complex type, or imaginary type invokes a complex function. Use of the macro with an argument of a decimal floating type results in undefined behavior.

[5] 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 (as in 7.25):

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

and the macros with **f***M*, **f***M***x**, **d***M*, or **d***M***x** prefix are:

f*M***add**	**f***M***xmul**	**d***M***fma**
f*M***sub**	**f***M***xdiv**	**d***M***sqrt**
f*M***mul**	**f***M***xfma**	**d***M***xadd**
f*M***div**	**f***M***xsqrt**	**d***M***xsub**
f*M***fma**	**d***M***add**	**d***M***xmul**
f*M***sqrt**	**d***M***sub**	**d***M***xdiv**
f*M***xadd**	**d***M***mul**	**d***M***xfma**
f*M***xsub**	**d***M***div**	**d***M***xsqrt**

All arguments are generic. If any argument is not real, use of the macro results in undefined behavior. The following specification uses the notation $type1 \subseteq type2$ to mean the values of *type*1 are a subset of (or the same as) the values of *type*2. The generic parameter type *T* for the function invoked by the macro is determined as follows:

— First apply the rules (for determining the corresponding real type of the generic parameters) in 7.25 for macros that do not round result to narrower type, using the usual arithmetic conversion rules in X.4.2, to obtain a preliminary type *P* for the generic parameters.

— If there exists a corresponding function whose generic parameters have type *P*, then *T* is *P*.

— Otherwise, *T* is determined from *P* and the macro prefix as follows:

— For prefix `f`: If P is a standard or binary floating type, then T is the first standard floating type in the list { `double`, `long double` } such that $P \subseteq T$, if such a type T exists. Otherwise (if no such type T exists or P is a decimal floating type), the behavior in undefined.

— For prefix `d`: If P is a standard or binary floating type, then T is `long double` if $P \subseteq$ `long double`. Otherwise (if $P \subseteq$ `long double` is false or P is a decimal floating type), the behavior in undefined.

— For prefix `f`*M*: If P is a standard or binary floating type, then T is `_Float`*N* for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported; otherwise T is `_Float`*N*`x` for minimum $N \geq M$ such that $P \subseteq T$, if such a type T is supported. Otherwise (if no such `_Float`*N* or `_Float`*N*`x` is supported or P is a decimal floating type), the behavior in undefined.

— For prefix `f`*M*`x`: If P is a standard or binary floating type, then T is `_Float`*N*`x` for minimum N > M such that $P \subseteq T$, if such a type T is supported; otherwise T is `_Float`*N* for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported. Otherwise (if no such `_Float`*N*`x` or `_Float`*N* is supported or P is a decimal floating type), the behavior in undefined.

— For prefix `d`*M*: If P is a decimal floating type, then T is `_Decimal`*N* for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported; otherwise T is `_Decimal`*N*`x` for minimum $N \geq M$ such that $P \subseteq T$. Otherwise (P is a standard or binary floating type), the behavior in undefined.

— For prefix `d`*M*`x`: If P is a decimal floating type, then T is `_Decimal`*N*`x` for minimum N > M such that $P \subseteq T$, if such a type T is supported; otherwise T is `_Decimal`*N* for minimum $N > M$ such that $P \subseteq T$, if such a type T is supported. Otherwise (P is a standard or binary floating type), the behavior in undefined.

EXAMPLE With the declarations

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <tgmath.h>
int n;
double d;
long double ld;
double complex dc;
_Float32x f32x;
_Float64 f64;
_Float64x f64x;
_Float128 f128;
_Float64x complex f64xc;
```

functions invoked by use of type-generic macros are shown in the following table, where *type1* $\subseteq$ *type2* means the values of *type*1 are a subset of (or the same as) the values of *type*2, and *type1* $\subset$ *type2* means the values of *type*1 are a strict subset of the values of *type*2:

macro use	invokes
`cos(f64xc)`	`ccosf64x`
`pow(dc, f128)`	`cpowf128`
`pow(f64, d)`	`powf64`

pow(d, f32x)	**pow**, the function, if **_Float32x** ⊆ **double**, else **powf32x** if **double** ⊂ **_Float32x**, else undefined
pow(f32, n)	**pow**, the function
pow(f32x, n)	**pow32x**

Macros that round result to narrower type …

fsub(d, ld)	**fsubl**
dsub(d, f32)	**dsubl**
fmul(dc, d)	undefined
ddiv(ld, f128)	**ddivl** if **_Float128** ⊆ **long double**, else undefined
f32add(f64x, f64)	**f32addf64x**
f32xsqrt(n)	**f32xsqrtf64**
f32mul(f128, f32x)	**f32mulf128** if **_Float32x** ⊆ **_Float128**, else **f32mulf32x** if **_Float128** ⊂ **_Float32x**, else undefined
f32fma(f32x, n, f32x)	**f32fmaf32x**
f32add(f32, f32)	**f32addf64**
f32xsqrt(f32)	**f32xsqrtf64x** Declaration shows **_Float64x** is supported.
f64div(f32x, f32x)	**f64divf128** if **_Float32x** ⊆ **_Float128**, else **f64divf64x**