## X.1 Introduction

### X.2 Types

- X.2.1 Interchange floating types
- X.2.2 Non-arithmetic interchange formats
- X.2.3 Extended floating types
- X.2.4 Classification of real floating types
- X.2.5 Complex types
- X.2.6 Imaginary types

### X.3 Characteristics in `<float.h>`

### X.4 Conversions

- X.4.1 Real floating and integer
- X.4.2 Usual arithmetic conversions
- X.4.3 Arithmetic and non-arithmetic formats

### X.5 Lexical elements

- X.5.1 Keywords
- X.5.2 Constants

### X.6 Expressions

### X.7 Declarations

### X.8 Identifiers in standard headers

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

### X.10 Floating-point environment

### X.11 Mathematics `<math.h>`

- X.11.1 Macros
- X.11.2 Function prototypes
- X.11.3 Encoding conversion functions
  - X.11.3.1 Encode and decode functions
  - X.11.3.2 Encoding-to-encoding conversion functions

### X.12 Numeric conversion functions in `<stdlib.h>`
X.12.2 String to floating
X.12.3 String from encoding
X.12.4 String to encoding
X.13 Type-generic macros <tgmath.h>

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

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

### Decimal interchange format parameters

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

*) 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

`_FloatN\), where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32

and types designated

`_DecimalN\), where \(N\) is 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 `_FloatN\), 10 for `_DecimalN\). 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 `_FloatN` 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 `_FloatN`; 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
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 `_FloatN` (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 `_DecimalN` 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, \( e_{\text{min}} = 3 - e_{\text{max}} \).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>binary32</th>
<th>binary64</th>
<th>binary128</th>
<th>decimal64</th>
<th>decimal128</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p ) digits ( \geq )</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>22</td>
<td>40</td>
</tr>
<tr>
<td>( e_{\text{max}} ) ( \geq )</td>
<td>1024</td>
<td>16384</td>
<td>65536</td>
<td>6145</td>
<td>24577</td>
</tr>
</tbody>
</table>

[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 _FloatN and _FloatNx are collectively called binary floating types.

[4] This subclause broadens decimal floating types to include the types _DecimalN and _DecimalNx 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:
    _DecimalN
    _DecimalNx
binary floating types:
    _FloatN
    _FloatNx
interchange floating types:
    _FloatN
    _DecimalN
extended floating types:
    _FloatNx
    _DecimalNx

[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 _FloatN and _FloatNx, there are complex types designated respectively as _FloatN _Complex and _FloatNx _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 _FloatN and _FloatNx, there are imaginary types designated respectively as _FloatN _Imaginary and _FloatNx _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 `_FloatN` 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 `_FloatN`, to the range and precision of `_FloatN`; evaluate all other operations and constants to the range and precision of the semantic type;

\[N + 1\], where `_FloatNx` 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 `_FloatNx`, to the range and precision of `_FloatNx`; 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 `_Decimal16`, to the range and precision of `_Decimal16`; 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 \(_\text{Decimal}N\) 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 \(_\text{Decimal}N\), to the range and precision of \(_\text{Decimal}N\); evaluate all other operations and constants to the range and precision of the semantic type;

\( N + 1 \), where \(_\text{Decimal}Nx\) 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 \(_\text{Decimal}Nx\), to the range and precision of \(_\text{Decimal}Nx\); evaluate all other operations and constants to the range and precision of the semantic type.

[4] This subclause also specifies \(<\text{float.h}>\) 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 \(\text{FLT}_N\) indicates the type \(_\text{Float}_N\) or the non-arithmetic binary interchange format of width \(N\). The prefix \(\text{FLT}_Nx\) indicates the type \(_\text{Float}_Nx\). The prefix \(\text{DEC}_N\) indicates the type \(_\text{Decimal}_N\) or the non-arithmetic decimal interchange format of width \(N\). The prefix \(\text{DEC}_Nx\) indicates the type \(_\text{Decimal}_Nx\). The type parameters \(p, e_{\text{max}}, \text{ and } e_{\text{min}}\) for extended floating types are for the extended floating type itself, not for the basic format that it extends.

[5] If \(_\text{STDC_WANT_IEC_60559_TYPES_EXT}_\) is defined (by the user) at the point in the code where \(<\text{float.h}>\) is first included, the following applies (X.8). For each interchange or extended floating type that the implementation provides, \(<\text{float.h}>\) shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, \(<\text{float.h}>\) shall not define the associated macros in the following list, except, the implementation shall define the macros \(\text{FLT}_N\_\text{DECIMAL}_\text{DIG}\) and \(\text{FLT}_N\_\text{DIG}\) 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 \(\text{FLT}_N\_\text{RADIX}\). 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, \(p\)

\(\text{FLT}_N\_\text{MANT}_\text{DIG}\)

\(\text{FLT}_Nx\_\text{MANT}_\text{DIG}\)

- number of digits in the coefficient, \(p\)

\(\text{DEC}_N\_\text{MANT}_\text{DIG}\)

\(\text{DEC}_Nx\_\text{MANT}_\text{DIG}\)
— 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, \( \left\lceil 1 + p \log_{10} 2 \right\rceil \)

\[
\texttt{FLTN\_DECIMAL\_DIG}
\]

\[
\texttt{FLTNX\_DECIMAL\_DIG}
\]

— 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, \( \left\lfloor (p - 1) \log_{10} 2 \right\rfloor \)

\[
\texttt{FLTN\_DIG}
\]

\[
\texttt{FLTNX\_DIG}
\]

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

\[
\texttt{FLTN\_MIN\_EXP}
\]

\[
\texttt{FLTNX\_MIN\_EXP}
\]

\[
\texttt{DECN\_MIN\_EXP}
\]

\[
\texttt{DECNX\_MIN\_EXP}
\]

— minimum negative integer such that \( 10 \) raised to that power is in the range of normalized floating-point numbers, \( \left\lceil \log_{10} 2^{e_{\min} - 1} \right\rceil \)

\[
\texttt{FLTN\_MIN\_10\_EXP}
\]

\[
\texttt{FLTNX\_MIN\_10\_EXP}
\]

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

\[
\texttt{FLTN\_MAX\_EXP}
\]

\[
\texttt{FLTNX\_MAX\_EXP}
\]

\[
\texttt{DECN\_MAX\_EXP}
\]

\[
\texttt{DECNX\_MAX\_EXP}
\]

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

\[
\texttt{FLTN\_MAX\_10\_EXP}
\]

\[
\texttt{FLTNX\_MAX\_10\_EXP}
\]

— maximum representable finite floating-point number, \( (1 - b^{-p}) b^{e_{\max}} \)

\[
\texttt{FLTN\_MAX}
\]

\[
\texttt{FLTNX\_MAX}
\]

\[
\texttt{DECN\_MAX}
\]

\[
\texttt{DECNX\_MAX}
\]

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

\[
\texttt{FLTN\_EPSILON}
\]

\[
\texttt{FLTNX\_EPSILON}
\]

\[
\texttt{DECN\_EPSILON}
\]

\[
\texttt{DECNX\_EPSILON}
\]
— minimum normalized positive floating-point number, \( b^{emin-1} \)

| FLT_MIN | FLT_MIN
| DEC_MIN | DEC_MIN

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

| FLT_TRUE_MIN | FLT_TRUE_MIN
| DEC_TRUE_MIN | DEC_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).

5 X.5 Lexical elements

5.1 Keywords

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

- `_FloatN`, where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
- `_Float32x`
- `_Float64x`
- `_Float128x`
- `_DecimalN`, where \(N\) is 96 or > 128 and a multiple of 32
- `_Decimal64x`
- `_Decimal128x`

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:

- \(fN\)
- \(FN\)
- \(fNx\)
- \(FNx\)
- \(dN\)
- \(DN\)
- \(dNx\)
- \(DNx\)

[3] A floating suffix \(dN\), \(DN\), \(dNx\), or \(DNx\) 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 \(fN\) or \(FN\), it has type `_FloatN`. If suffixed by \(fNx\) or \(FNx\), it has type `_FloatNx`. If suffixed by \(dN\) or \(DN\), it has type `_DecimalN`. If suffixed by \(dNx\) or \(DNx\), it has type `_DecimalNx`.

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

[7] **NOTE** For \(N = 32, 64, \) and 128, the suffixes \(dN\) and \(DN\) in this subclause for constants of type `_DecimalN` are equivalent alternatives to the suffixes `df`, `dd`, `dl`, `DF`, `DD`, and `DL` in 6.4.4.2 for the same types.

6 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:

- _FloatN, where N is 16, 32, 64, or ≥ 128 and a multiple of 32
- _Float32x
- _Float64x
- _Float128x
- _DecimalN, where N is 96 or > 128 and a multiple of 32
- _Decimal64x
- _Decimal128x

[2] The type specifiers _FloatN (where N is 16, 32, 64, or ≥ 128 and a multiple of 32), _Float32x, _Float64x, _Float128x, _DecimalN (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:

<table>
<thead>
<tr>
<th></th>
<th>_FloatN, where N is 16, 32, 64 or ≥ 128 and a multiple of 32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Float32x</td>
</tr>
<tr>
<td></td>
<td>_Float64x</td>
</tr>
<tr>
<td></td>
<td>_Float128x</td>
</tr>
<tr>
<td></td>
<td>_DecimalN, where N is 96 or &gt; 128 and a multiple of 32</td>
</tr>
<tr>
<td></td>
<td>_Decimal64x</td>
</tr>
<tr>
<td></td>
<td>_Decimal128x</td>
</tr>
</tbody>
</table>

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 fN and fNx suffixes, which are corresponding functions whose parameters and return values have corresponding real types _FloatN and _FloatNx.

[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 fN and fNx.

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`, `acosfN`, and `acosfNx` as well as `acos`). The `fMencfN`, `strfromencfN`, and `strtoencfN` 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 `acosdN` and `acosdNx`). The `dMencbindN`, `dMencdecdN`, `strfromencbindN`, `strfromencdecdN`, `strtoencbindN`, and `strtoencdecdN` 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 `fN`, `fNx`, `dN`, and `dNx` suffixes, which are corresponding functions whose parameters, return values, or both are of types `_FloatN`, `_FloatNx`, `_DecimalN`, and `_DecimalNx`, 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

```c
float_t
double_t
```

in 7.12 are included the type

```c
long_double_t
```

and for each supported type `_FloatN`, the type

```c
_FloatN_t
```

and for each supported type `_DecimalN`, the type

```c
_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`;

— `_FloatN_t` has at least the range and precision of `_FloatM_t` if \( N > M \);

— `_DecimalN_t` has at least the range and precision of `_DecimalM_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

<table>
<thead>
<tr>
<th>Type</th>
<th>IEC 60559 format</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Float16</td>
<td>binary16</td>
</tr>
<tr>
<td>float, _Float32</td>
<td>binary32</td>
</tr>
<tr>
<td>double, _Float64, _Float32x</td>
<td>binary64</td>
</tr>
<tr>
<td>long double, _Float64x</td>
<td>80-bit binary64-extended</td>
</tr>
<tr>
<td>_Float128</td>
<td>binary128</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>_t_type ( m )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Float16_t</td>
<td>float</td>
<td>double</td>
<td>long double</td>
<td>_Float32</td>
<td>_Float64</td>
<td>_Float128</td>
<td>_Float32x</td>
<td>_Float64x</td>
</tr>
<tr>
<td>float_t</td>
<td>float</td>
<td>double</td>
<td>long double</td>
<td>float</td>
<td>_Float64</td>
<td>_Float128</td>
<td>_Float32x</td>
<td>_Float64x</td>
</tr>
<tr>
<td>_Float32_t</td>
<td>_Float32</td>
<td>double</td>
<td>long double</td>
<td>_Float32</td>
<td>_Float64</td>
<td>_Float128</td>
<td>_Float32x</td>
<td>_Float64x</td>
</tr>
<tr>
<td>double_t</td>
<td>double</td>
<td>double</td>
<td>long double</td>
<td>double</td>
<td>double</td>
<td>_Float128</td>
<td>double</td>
<td>_Float64x</td>
</tr>
<tr>
<td>_Float64_t</td>
<td>_Float64</td>
<td>long double</td>
<td>long double</td>
<td>_Float64</td>
<td>_Float64</td>
<td>_Float128</td>
<td>_Float64</td>
<td>_Float64x</td>
</tr>
<tr>
<td>long double_t</td>
<td>long double</td>
<td>long double</td>
<td>long double</td>
<td>long double</td>
<td>long double</td>
<td>_Float128</td>
<td>long double</td>
<td>_Float128</td>
</tr>
<tr>
<td>_Float128_t</td>
<td>_Float128</td>
<td>_Float128</td>
<td>_Float128</td>
<td>_Float128</td>
<td>_Float128</td>
<td>_Float128</td>
<td>_Float128</td>
<td></td>
</tr>
</tbody>
</table>

### X.11.1 Macros

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

[2] The macros

```
HUGE_VAL_F, HUGE_VAL_FN
HUGE_VAL_D, HUGE_VAL_DN
HUGE_VAL_FNX, HUGE_VAL_DNX
```

expand to constant expressions of types `_FloatN`, `_DecimalN`, `_FloatNx`, and `_DecimalNx`, respectively, representing positive infinity.
The signaling NaN macros

```c
SNANFN
SNANDN
SNANFX
SNANDNX
```

expand to constant expressions of types \_Float\_N, \_Decimal\_N, \_Float\_Nx, and \_Decimal\_Nx, 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.]

The macros

```c
FP_FAST_FMAFN
FP_FAST_FMADN
FP_FAST_FMAFX
FP_FAST_FMADNX
```

are, respectively, \_Float\_N, \_Decimal\_N, \_Float\_Nx, and \_Decimal\_Nx analogues of \_FP_FAST_FMA.\_N.

The macros in the following lists are interchange and extended floating type analogues of \_FP_FAST_FADD, \_FP_FAST_FADDL, \_FP_FAST_DADDL, etc.

For \( M < N \), the macros

```c
FP_FAST_FMAFDFN
FP_FAST_FMSUBFBN
FP_FAST_FMFULFN
FP_FAST_FMIDFVN
FP_FAST_FMFMAFN
FP_FAST_FMSQRTFN
FP_FAST_DMAADDN
FP_FAST_DMSUBD
FP_FAST_DMULDN
FP_FAST_DDIVD
FP_FAST_DMFPADN
FP_FAST_DMSQRTD
```
[7] For $M \leq N$, the macros

```
FP_FAST_FMADDFNX
FP_FAST_FMSUBFNX
FP_FAST_FMULFNX
FP_FAST_PMDIVFNX
FP_FAST_FMFMFNX
FP_FAST_MSQRTFNX
FP_FAST_DMADDNX
FP_FAST_DMSUBDNX
```

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_FMXADDFN
FP_FAST_FMXSUBFN
FP_FAST_FMXMULFN
FP_FAST_PMXDIVFN
FP_FAST_FMXFMFN
FP_FAST_MSQRTFN
FP_FAST_DMXADDN
FP_FAST_DMXSUBDN
```

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_FMXADDFNX
FP_FAST_FMXSUBFNX
FP_FAST_FMXMULFNX
FP_FAST_PMXDIVFNX
FP_FAST_FMXFMFNX
FP_FAST_MSQRTFNX
FP_FAST_DMXADDNX
FP_FAST_DMXSUBDNX
```

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(DecimalN 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);

_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 sinhN(_DecimalN x);
.DecimalNx sinhNx(_DecimalNx x);
7.12.6 Exponential and logarithmic functions

_FloatN tanhfN(_FloatN x);
_FloatN x tanhfN(_FloatN x);
_DecimalN tanhdN(_DecimalN x);
_DecimalN x tanhdNx(_DecimalN x);

_FloatN expfN(_FloatN x);
_FloatN x expfNx(_FloatN x);
_DecimalN expdN(_DecimalN x);
_DecimalN x expdNx(_DecimalN x);

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

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

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

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

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

_FloatN frexpN(_FloatN value, int *exp);
_FloatN x frexpNx(_FloatN value, int *exp);
_DecimalN frexpN(_DecimalN value, int *exp);
_DecimalN x frexpNx(_DecimalN value, int *exp);

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

_FloatN ldexpfN(_FloatN value, int exp);
_FloatN x ldexpfNx(_FloatN value, int exp);
_DecimalN ldexpdN(_DecimalN value, int exp);
_DecimalN x ldexpdNx(_DecimalN 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);
_FloatN logfNx(_FloatNx x);
_DecimalN logdN(_DecimalN x);
_DecimalN logdNx(_DecimalNx x);

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

_FloatN log1pfN(_FloatN x);
_FloatN log1pfNx(_FloatNx x);
_FloatN log1fN(_FloatN x);
_FloatN log1fNx(_FloatNx x);
_DecimalN log1dN(_DecimalN x);
_DecimalN log1dNx(_DecimalNx x);

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

_FloatN log2p1fN(_FloatN x);
_FloatN log2p1fNx(_FloatNx x);
_FloatN log21fN(_FloatN x);
_FloatN log21fNx(_FloatNx x);
_DecimalN log21dN(_DecimalN x);
_DecimalN log21dNx(_DecimalNx x);

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

_FloatN modffN(_FloatN x, _FloatN *iptr);
_FloatN modffNx(_FloatNx x, _FloatNx *iptr);
_DecimalN modfDN(_DecimalN x, _DecimalN *iptr);
_DecimalN modfDx(_DecimalN x, _DecimalNx *iptr);

_FloatN scalbnfN(_FloatN value, int exp);
_FloatN scalbnfx(_FloatNx value, int exp);
_DecimalN scalbnfD(_DecimalN value, int exp);
_DecimalN scalbnfxD(_DecimalNx value, int exp);
7.12.7 Power and absolute-value functions

_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);

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

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

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

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

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

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

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

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

_FloatN rsqrfN(_FloatN x);
_FloatNx rsqrfNx(_FloatNx x);
_DecimalN rsqrtdN(_DecimalN x);
_DecimalNx rsqrtdNx(_DecimalN x x);
7.12.8 Error and gamma functions

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

_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);
<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>llrintf</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llrintfx</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llrintd</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llrintdx</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundf</code></td>
<td>Long double value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundfx</code></td>
<td>Long double value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundd</code></td>
<td>Long double value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>rounddx</code></td>
<td>Long double value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>lroundf</code></td>
<td>Long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>lroundfx</code></td>
<td>Long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>lroundd</code></td>
<td>Long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>lrounddx</code></td>
<td>Long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llroundf</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llroundfx</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llroundd</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>llrounddx</code></td>
<td>Long long int value of <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundevenf</code></td>
<td>Round <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundevenfx</code></td>
<td>Round <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundevend</code></td>
<td>Round <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>roundevendx</code></td>
<td>Round <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>truncf</code></td>
<td>Truncate <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>trunclfx</code></td>
<td>Truncate <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>truncld</code></td>
<td>Truncate <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>truncldx</code></td>
<td>Truncate <code>x</code> to nearest integer.</td>
</tr>
<tr>
<td><code>fromfpf</code></td>
<td>Convert float to integer.</td>
</tr>
<tr>
<td><code>fromfpfx</code></td>
<td>Convert float to integer.</td>
</tr>
<tr>
<td><code>fromfpd</code></td>
<td>Convert decimal to integer.</td>
</tr>
<tr>
<td><code>fromfpdx</code></td>
<td>Convert decimal to integer.</td>
</tr>
<tr>
<td><code>ufromfpf</code></td>
<td>Convert float to unsigned integer.</td>
</tr>
<tr>
<td><code>ufromfpfx</code></td>
<td>Convert float to unsigned integer.</td>
</tr>
<tr>
<td><code>ufromfpd</code></td>
<td>Convert decimal to unsigned integer.</td>
</tr>
<tr>
<td><code>ufromfpdx</code></td>
<td>Convert decimal to unsigned integer.</td>
</tr>
<tr>
<td><code>fromfpxf</code></td>
<td>Convert float to integer.</td>
</tr>
<tr>
<td><code>fromfpxfx</code></td>
<td>Convert float to integer.</td>
</tr>
<tr>
<td><code>fromfpxd</code></td>
<td>Convert decimal to integer.</td>
</tr>
<tr>
<td><code>fromfpxdx</code></td>
<td>Convert decimal to integer.</td>
</tr>
</tbody>
</table>
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);
Floating multiply-add

- FloatN fmafN(_FloatN x, _FloatN y, _FloatN z);
  _FloatN fmfNx(_FloatN x, _FloatN y, _FloatN z);
  _DecimalN fmfdN(_DecimalN x, _DecimalN y, _DecimalN z);

Functions that round result to narrower type

- FloatM fMaddfN(_FloatN x, _FloatN y);  // M < N
  _FloatM fMfN(_FloatN x, _FloatN y);     // M <= N
  _FloatM fMxaddfN(_FloatN x, _FloatN y); // M < N
  _DecimalM dMfddN(_DecimalN x, _DecimalN y);  // M < N
  _DecimalM dMfddNx(_DecimalN x, _DecimalN y); // M <= N
  _DecimalM dMxaddfN(_DecimalN x, _DecimalN y); // M < N
  _DecimalM dMxaddfNx(_DecimalN x, _DecimalN y); // M <= N
  _DecimalM dMsubfN(_FloatN x, _FloatN y);  // M < N
  _FloatM fMfN(_FloatN x, _FloatN y);     // M <= N
7.12.15 Quantum and quantum exponent functions

_Decimal N quantizedN(_Decimal N x, _Decimal N y); // M < N
_Decimal Nx quantizedNx(_DecimalNx x, _DecimalNx y);

_Bool samequantumdN(_Decimal N x, _Decimal N y);
_Bool samequantumdNx(_DecimalNx x, _DecimalNx y);

_Decimal N quantumdN(_Decimal N x);
_Decimal Nx quantumdNx(_DecimalNx x);
long long int llquantexpd(_DecimalN x);
long long int llquantexpdN(_DecimalN x);

7.12.16 Decimal re-encoding functions

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

```c
void encodedbindN(unsigned char * restrict encptr,
                   const _DecimalN * restrict xptr);
void decodedbindN(_DecimalN * restrict xptr,
                  const unsigned char * restrict encptr);
```

F.10.12 Total order functions

```c
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);
```

F.10.13 Payload functions

```c
_FloatN getpayloadfN(const _FloatN *x);
_FloatNx getpayloadfnx(const _FloatNx *x);
_DecimalN getpayloaddN(const _DecimalN *x);
_DecimalNx getpayloaddnx(const _DecimalNx *x);
```

```c
int setpayloadfN(_FloatN *res, _FloatN pl);
int setpayloadfnx(_FloatNx *res, _FloatNx pl);
int setpayloaddN(_DecimalN *res, _DecimalN pl);
int setpayloaddnx(_DecimalNx *res, _DecimalNx pl);
```

40 [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 \( x \times 2^{\text{exp}} \).

45 [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 `encodefN` functions

Synopsis

```
#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 `encodefN` 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


X.11.3.1.2 The `decodefN` functions

Synopsis

```
#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 `decodefN` 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 `_FloatN`, 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


X.11.3.2 Encoding-to-encoding conversion functions

[1] An implementation shall declare an `fMencfN` 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 `dMenceddN` and `dMencbindN` 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 `fMencfN` 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 `encNptr` 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 `encMptr`. 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:

```c
#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);
deodef32(&f32, b32);
f = f32;
...
```
X.11.3.2.2 The dMencdecN and dMencbindN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <math.h>

void dMencdec(N(unsigned char encMptr[restrict static M/8],
const unsigned char encNptr[restrict static N/8]);
void dMencbind(N(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 dMencdecN functions convert between formats using the encoding scheme based on decimal encoding of the significand. The dMencbindN 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 encNptr 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 encMptr. 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 strtof 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 strfromfx(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

```c
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

```c
_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

```c
_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 `strtodN` and `strtodNx` functions (and hence the `strtoencN` and `strtoencN` 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 $M = \left\lceil (P-1)/4 \right\rceil + 1$ 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:

```c
#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);
45 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

[-0xh...hptd, 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 strfromencfN 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 strfromencdecN and strfromencbind 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 strfromencfN 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 strfromencfN functions are similar to the strfromfN 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 strfromencfN functions return the same values as corresponding strfromfN functions.

X.12.3.2 The strfromencdecN and strfromencbindN functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_TYPES_EXT__
#include <stdlib.h>
int strfromencdecN(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 strfromencdecN functions are similar to the strfromdN 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 functions are similar to the strfromdN 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 `strfromencdecN` and `strfromencbindN` functions return the same values as corresponding `strfromd` functions.

X.12.4 String to encoding

[1] An implementation shall declare the `strtoencfN` 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 `strtoencdecN` and `strtoencbindN` 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 `strtoencfN` 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 endpstr);

Description

[2] The `strtoencfN` functions are similar to the `strtofN` 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 `strtoencdecN` and `strtoencbindN` functions

Synopsis

[1] `#define __STDC_WANT_IEC_60559_TYPES_EXT__`

   `#include <stdlib.h>`

   `void strtoencdecN(unsigned char encptr[restrict static N/8],
                     const char * restrict nptr, char ** restrict endpstr);`

   `void strtoencbindN(unsigned char encptr[restrict static N/8],
                      const char * restrict nptr, char ** restrict endpstr);`

Description

[2] The `strtoencdecN` and `strtoencbindN` functions are similar to the `strtodN` 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 `strtoencdecN` functions produce an encoding in the encoding scheme based on decimal encoding of the significand. The `strtoencbindN` 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  
dadd  
fsub  
dsub
```

```
fmul  
dmul  
fdiv  
ddiv
```

```
ffma  
dfma  
fsqrt 
dsqrt
```

and the macros with fM, fMx, dM, or dMx prefix are:

```
fMadd  
fMsub  
fMmul  
fMddiv  
fMfma  
fMsqrt 
fMxadd  
fMxsub 
fMxmul  
fMxfma  
fMxsqrt
```

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 ⊆ type2 to mean the values of type1 are a subset of (or the same as) the values of type2. 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 fM: 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$Nx$ 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$Nx$ is supported or $P$ is a decimal floating type), the behavior in undefined.

— For prefix fMx: If $P$ is a standard or binary floating type, then $T$ is _Float$Nx$ for minimum $N > M$ such that $P \subseteq T$, if such a type $T$ is supported; otherwise $T$ is _Float$N$ 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$Nx$ is supported or $P$ is a decimal floating type), the behavior in undefined.

— For prefix dM: 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$Nx$ for minimum $N \geq M$ such that $P \subseteq T$. Otherwise (if $P$ is a standard or binary floating type), the behavior in undefined.

— For prefix dMx: If $P$ is a decimal floating type, then $T$ is _Decimal$Nx$ 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 (if $P$ is a standard or binary floating type), the behavior in undefined.

**EXAMPLE** With the declarations

```c
#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;
_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 type1 are a subset of (or the same as) the values of type2, and type1 \subset type2 means the values of type1 are a strict subset of the values of type2:

<table>
<thead>
<tr>
<th>macro use</th>
<th>invokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos(f64xc)</td>
<td>ccosf64x</td>
</tr>
<tr>
<td>pow(dc, f128)</td>
<td>cpowf128</td>
</tr>
<tr>
<td>pow(f64, d)</td>
<td>powf64</td>
</tr>
</tbody>
</table>
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) dsbl

fmul(dc, d) undefined

ddiv(ld, f128) ddivl if _Float128 ⊆ long double, else
undefined

f32add(f64x, f64) f32addf64x

f32sqrtf32x f32xsqrtf64

f32mul(f128, f32x) f32mulf128 if _Float32x ⊆ _Float128, else
f32mulf32x if _Float128 ⊆ _Float32x, else
undefined

f32fma(f32x, n, f32x) f32fmaf32x

f32add(f32, f32) f32addf64

f32sqrtf32x Declaration shows _Float64x is supported.

f64div(f32x, f32x) f64divf128 if _Float32x ⊆ _Float128, else
f64divf64x