Annex X
(normative)
IEC 60559 interchange and extended types

X.1 Introduction

[1] This annex extends programming language C to include types with the arithmetic interchange and extended floating-point formats specified in ISO/IEC/IEEE 60559:2011, and to include 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 all-20190708:

In 6.10.8.3#1, add:

20 __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, identified by their width, which can be used for the exchange of floating-point data between implementations. The two tables below give parameters for the IEC 60559 interchange formats.
### 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>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>(e_{\text{max}}), maximum exponent (e)</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
<td>(2^{(N-1)} - 1)</td>
</tr>
</tbody>
</table>

**Encoding parameters**

<table>
<thead>
<tr>
<th>bias, (E-e)</th>
<th>15</th>
<th>127</th>
<th>1023</th>
<th>16383</th>
<th>(e_{\text{max}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(w), exponent field width in bits</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>15</td>
<td>(\text{round}(4 \times \log_2(N)) - 13)</td>
</tr>
<tr>
<td>(t), trailing significand field width in bits</td>
<td>10</td>
<td>23</td>
<td>52</td>
<td>112</td>
<td>(N - w - 1)</td>
</tr>
<tr>
<td>(N), storage width in bits</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>(1 + w + t)</td>
</tr>
</tbody>
</table>

The function \(\text{round}()\) in the table above rounds to the nearest integer. For example, binary256 would have \(p = 237\) and \(e_{\text{max}} = 262143\).

### 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>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>(e_{\text{max}}), maximum exponent (e)</td>
<td>96</td>
<td>384</td>
<td>6144</td>
<td>(3 \times 2^{(N/16 + 3)})</td>
</tr>
</tbody>
</table>

**Encoding parameters**

<table>
<thead>
<tr>
<th>bias, (E-e)</th>
<th>101</th>
<th>398</th>
<th>6176</th>
<th>(e_{\text{max}} + p - 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(W+5), combination field width in bits</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>(N/16 + 9)</td>
</tr>
<tr>
<td>(t), trailing significand field width in bits</td>
<td>20</td>
<td>50</td>
<td>110</td>
<td>(15 \times N/16 - 10)</td>
</tr>
<tr>
<td>(N), storage width in bits</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>(1 + 5 + w + t)</td>
</tr>
</tbody>
</table>

For example, decimal256 would have \(p = 70\) and \(e_{\text{max}} = 1572864\).

[2] Types designated

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

10 and types designated

\_Decimal\(N\), where \(N \geq 32\) 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.

[3] 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 binary interchange floating types; the set of such types supported is implementation-defined.

[4] An implementation that defines \_STDC\_IEC\_60559\_DFP shall provide the types \_Decimal32, \_Decimal64, and \_Decimal128. If the implementation also defines \_STDC\_IEC\_60559\_TYPES, it may provide other decimal interchange floating types; the set of such types supported is implementation-defined.

### X.2.2 Non-arithmetic interchange formats

[1] Providing an interchange floating type entails supporting it as an IEC 60559 arithmetic format. An implementation supports IEC 60559 non-arithmetic interchange formats by providing the associated encoding-to-encoding conversion functions (X.11.3.2), string-from-encoding functions (X.12.3), and string-to-encoding functions (X.12.4).

[2] An implementation that defines \_STDC\_IEC\_60559\_BFP and \_STDC\_IEC\_60559\_TYPES shall support the IEC 60559 binary16 format, at least as a non-arithmetic interchange format. Otherwise, the set of non-arithmetic interchange 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. The table below gives the minimum values of these parameters:

<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 ≥</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>22</td>
<td>40</td>
</tr>
<tr>
<td>(e_{max}) ≥</td>
<td>1023</td>
<td>16383</td>
<td>65535</td>
<td>6144</td>
<td>24576</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. 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, which may have the same set of values as double, and may provide any of the other two binary extended floating types. An implementation that defines \_STDC\_IEC\_60559\_DFP and \_STDC\_IEC\_60559\_TYPES shall provide: \_Decimal64x, which may have the same set of values as \_Decimal128, and may provide \_Decimal128x. Which (if any) of the optional extended floating types are provided is implementation-defined.

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 \_Decimal\(32\) type.
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`. This subclause broadens the definition of decimal floating types
and defines binary floating types to be collective names for types for all the appropriate IEC 60559
arithmetic formats. Note that standard floating types (which have an implementation-defined radix)
are not included in either decimal floating types (which all have radix 10) or binary floating types
(which all have radix 2).

[2] The real floating types are broadened to include all interchange floating types and extended floating
types, as well as standard floating types.

[3] The interchange floating types designated `_FloatN` and the extended floating types designated
`_FloatNx` are collectively called the binary floating types. The interchange floating types designated
`_DecimalN` and the extended floating types designated `_DecimalNx` are collectively called the
decimal floating types. Note that binary floating types and the decimal floating types are real floating
types.

[4] Thus real floating types are classified as follows:

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

binary floating types:
`_FloatN`
`_FloatNx`

decimal floating types:
`_DecimalN`
`_DecimalNx`

X.2.5 Complex types

[1] This subclause broadens the C complex types (6.2.5) to also include similar types whose

30 corresponding real parts are binary floating types. For the interchange floating types `_FloatN`, and
the extended floating types `_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

35 corresponding real parts are binary floating types. For the interchange floating types `_FloatN` and
the extended floating types `_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 specifies new `<float.h>` macros, analogous to the macros for standard floating
types, that characterize this annex’s binary floating types in terms of the model presented in 5.2.4.2.2.
It generalizes the specification of characteristics in 5.2.4.2.3 of the three decimal floating types to include this annex's decimal floating types. The prefix \texttt{FLT\_} indicates a binary interchange floating type or a non-arithmetic binary interchange format of width $N$. The prefix \texttt{FLT\_NX\_} indicates a binary extended floating type that extends a basic format of width $N$. The prefix \texttt{DEC\_} indicates a decimal interchange floating type of width $N$. The prefix \texttt{DEC\_NX\_} indicates a decimal extended floating type that extends a basic format of width $N$. The type parameters $p$, $e_{\text{max}}$ and $e_{\text{min}}$ for extended floating types are for the extended floating type itself, not for the basic format that it extends. For each interchange or extended floating type that the implementation provides, \texttt{<float.h>} shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, \texttt{<float.h>} shall not define the associated macros in the following list, except, the implementation shall define the macros \texttt{FLT\_DECIMAL\_DIG} and \texttt{FLT\_DIG} if it supports IEC 60559 non-arithmetic binary interchange formats of width $N$ by providing the encoding-to-encoding conversion functions (X.11.3.2) in \texttt{<math.h>} and the string-from-encoding (X.12.3) and string-to-encoding (X.12.4) functions in \texttt{<stdlib.h>}.  

[2] If \texttt{FLT\_RADIX} is 2, the value of the macro \texttt{FLT\_EVAL\_METHOD} (5.2.4.2.2) characterizes the use of evaluation formats for standard floating types and for binary interchange and extended floating types:

\begin{itemize}
  \item [-1] indeterminable;
  \item [0] evaluate all operations and constants, whose semantic type has at most the range and precision of \texttt{float}, to the range and precision of \texttt{float}; evaluate all other operations and constants to the range and precision of the semantic type;
  \item [1] evaluate operations and constants, whose semantic type has at most the range and precision of \texttt{double}, to the range and precision of \texttt{double}; evaluate all other operations and constants to the range and precision of the semantic type;
  \item [2] evaluate operations and constants, whose semantic type has at most the range and precision of \texttt{long double}, to the range and precision of \texttt{long double}; evaluate all other operations and constants to the range and precision of the semantic type;
  \item [N, where \texttt{\_Float\_N\_\_} is a supported interchange floating type]
    evaluate operations and constants, whose semantic type has at most the range and precision of the \texttt{\_Float\_N\_\_} type, to the range and precision of the \texttt{\_Float\_N\_\_} type; evaluate all other operations and constants to the range and precision of the semantic type;
  \item [N + 1, where \texttt{\_Float\_NX\_\_} is a supported extended floating type]
    evaluate operations and constants, whose semantic type has at most the range and precision of the \texttt{\_Float\_NX\_\_} type, to the range and precision of the \texttt{\_Float\_NX\_\_} type; evaluate all other operations and constants to the range and precision of the semantic type.
\end{itemize}

If \texttt{FLT\_RADIX} is not 2, the use of evaluation formats for operations and constants of binary interchange and extended floating types is implementation-defined.

[3] The implementation-defined value of the macro \texttt{DEC\_EVAL\_METHOD} (5.2.4.2.3) characterizes the use of evaluation formats for decimal interchange and extended floating types:

\begin{itemize}
  \item [-1] indeterminable;
  \item [0] evaluate all operations and constants just to the range and precision of the type;
  \item [1] evaluate operations and constants, whose semantic type has at most the range and precision of the \texttt{\_Decimal16\_4\_\_} type, to the range and precision of the \texttt{\_Decimal16\_4\_\_} type.
\end{itemize}
type; evaluate all other operations and constants to the range and precision of the semantic type;

2 evaluate operations and constants, whose semantic type has at most the range and precision of the _Decimal128 type, to the range and precision of the _Decimal128 type; evaluate all other operations and constants to the range and precision of the semantic type;

\[ N, \text{where } _\text{Decimal} N \text{ is a supported interchange floating type} \]
evaluate operations and constants, whose semantic type has at most the range and precision of the _DecimalN type, to the range and precision of the _DecimalN type; evaluate all other operations and constants to the range and precision of the semantic type;

\[ N + 1, \text{where } _\text{Decimal} N x \text{ is a supported extended floating type} \]
evaluate operations and constants, whose semantic type has at most the range and precision of the _DecimalNx type, to the range and precision of the _DecimalNx type; evaluate all other operations and constants to the range and precision of the semantic type;

[4] 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, \( b (= 2 \text{ for binary, 10 for decimal}) \)

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, \( p \)

\[
\begin{align*}
\text{FLT}_N \_MANT\_DIG \\
\text{FLT}_N X \_MANT\_DIG
\end{align*}
\]

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

\[
\begin{align*}
\text{DEC}_N \_MANT\_DIG \\
\text{DEC}_N X \_MANT\_DIG
\end{align*}
\]

— 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, \( \left\lceil 1 + p \log_{10} 2 \right\rceil \)

\[
\begin{align*}
\text{FLT}_N \_DECIMAL\_DIG \\
\text{FLT}_N X \_DECIMAL\_DIG
\end{align*}
\]

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

\[
\begin{align*}
\text{FLT}_N \_DIG \\
\text{FLT}_N X \_DIG
\end{align*}
\]
— minimum negative integer such that the radix raised to one less than that power is a normalized floating-point number, $e_{\text{min}}$

\[
\begin{align*}
\text{FLT}_N\_\text{MIN}\_\text{EXP} \\
\text{FLT}_N\_\text{MIN}\_\text{EXP} \\
\text{DEC}_N\_\text{MIN}\_\text{EXP} \\
\text{DEC}_N\_\text{MIN}\_\text{EXP}
\end{align*}
\]

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $[\log_{10} 2^{e_{\text{min}-1}}]$ 

\[
\begin{align*}
\text{FLT}_N\_\text{MIN}\_10\_\text{EXP} \\
\text{FLT}_N\_\text{MIN}\_10\_\text{EXP}
\end{align*}
\]

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

\[
\begin{align*}
\text{FLT}_N\_\text{MAX}\_\text{EXP} \\
\text{FLT}_N\_\text{MAX}\_\text{EXP} \\
\text{DEC}_N\_\text{MAX}\_\text{EXP} \\
\text{DEC}_N\_\text{MAX}\_\text{EXP}
\end{align*}
\]

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

\[
\begin{align*}
\text{FLT}_N\_\text{MAX}\_10\_\text{EXP} \\
\text{FLT}_N\_\text{MAX}\_10\_\text{EXP}
\end{align*}
\]

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

\[
\begin{align*}
\text{FLT}_N\_\text{MAX} \\
\text{FLT}_N\_\text{MAX} \\
\text{DEC}_N\_\text{MAX} \\
\text{DEC}_N\_\text{MAX}
\end{align*}
\]

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

\[
\begin{align*}
\text{FLT}_N\_\text{EPSILON} \\
\text{FLT}_N\_\text{EPSILON} \\
\text{DEC}_N\_\text{EPSILON} \\
\text{DEC}_N\_\text{EPSILON}
\end{align*}
\]

— minimum normalized positive floating-point number, $b^{e_{\text{min}-1}}$

\[
\begin{align*}
\text{FLT}_N\_\text{MIN} \\
\text{FLT}_N\_\text{MIN} \\
\text{DEC}_N\_\text{MIN} \\
\text{DEC}_N\_\text{MIN}
\end{align*}
\]

— minimum positive subnormal floating-point number, $b^{e_{\text{min}-p}}$

\[
\begin{align*}
\text{FLT}_N\_\text{TRUE}\_\text{MIN} \\
\text{FLT}_N\_\text{TRUE}\_\text{MIN} \\
\text{DEC}_N\_\text{TRUE}\_\text{MIN} \\
\text{DEC}_N\_\text{TRUE}\_\text{MIN}
\end{align*}
\]
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 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.

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

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.

Otherwise, if both operands have floating types, the operand, whose set of values of its corresponding real type is a (proper) 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 one operand has a floating type, the other operand is converted to the corresponding real type of the operand of floating type.

**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 - 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.3.2) and decode functions (X.11.3.1.2, 7.12.16.2, 7.12.16.4).

— 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).

— 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:

- _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_

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:

- 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_.

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] 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.

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

[5] 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 $\geq 128$ 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 $\geq 128$ 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:

- `- _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`
- `- _FloatN _Complex`, where $N$ is 16, 32, 64, or $\geq 128$ and a multiple of 32
- `- _Float32x _Complex`
- `- _Float64x _Complex`
- `- _Float128x _Complex`

X.8 Identifiers in standard headers

[1] The new identifiers added to C library headers by this annex are defined or declared by their respective headers only if `__STDC_WANT_IEC_60559_TYPES_EXT__` is defined as a macro at the point in the source file where the appropriate header is first included. The following subclauses list these identifiers for each applicable standard header.

X.8.1 `<float.h>`

[1] The following identifiers are defined only if `__STDC_WANT_IEC_60559_TYPES_EXT__` is defined as a macro at the point in the source file where `<float.h>` is first included:
for supported types _FloatN:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT N MANT DIG</td>
<td>FLT N MIN 10 EXP</td>
<td>FLT N EPSILON</td>
</tr>
<tr>
<td>FLT N DECIMAL DIG</td>
<td>FLT N MAX EXP</td>
<td>FLT N MIN</td>
</tr>
<tr>
<td>FLT N DIG</td>
<td>FLT N MAX 10 EXP</td>
<td>FLT N TRUE_MIN</td>
</tr>
<tr>
<td>FLT N MIN EXP</td>
<td>FLT N MAX</td>
<td></td>
</tr>
</tbody>
</table>

for IEC 60559 non-arithmetic binary interchange formats of width N:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT N DECIMAL DIG</td>
<td>FLT N DIG</td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT Nx MANT DIG</td>
<td>FLT Nx MIN 10 EXP</td>
<td>FLT Nx EPSILON</td>
</tr>
<tr>
<td>FLT Nx DECIMAL DIG</td>
<td>FLT Nx MAX EXP</td>
<td>FLT Nx MIN</td>
</tr>
<tr>
<td>FLT Nx DIG</td>
<td>FLT Nx MAX 10 EXP</td>
<td>FLT Nx TRUE_MIN</td>
</tr>
<tr>
<td>FLT Nx MIN EXP</td>
<td>FLT Nx MAX</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _DecimalN, where N ≠ 32, 64, and 128:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC N MANT DIG</td>
<td>DEC N MAX</td>
<td>DEC N TRUE_MIN</td>
</tr>
<tr>
<td>DEC N MIN EXP</td>
<td>DEC N EPSILON</td>
<td></td>
</tr>
<tr>
<td>DEC N MAX EXP</td>
<td>DEC N MIN</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _DecimalNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC N MANT DIG</td>
<td>DEC N MAX</td>
<td>DEC N TRUE_MIN</td>
</tr>
<tr>
<td>DEC N MIN EXP</td>
<td>DEC N EPSILON</td>
<td></td>
</tr>
<tr>
<td>DEC N MAX EXP</td>
<td>DEC N MIN</td>
<td></td>
</tr>
</tbody>
</table>

X.8.2 <complex.h>

[1] The following identifiers are declared or defined only if __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where <complex.h> is first included:

for supported types _FloatN:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfN</td>
<td>catanhfN</td>
<td>csqrtfN</td>
</tr>
<tr>
<td>casinfN</td>
<td>ccoshfN</td>
<td>cargfN</td>
</tr>
<tr>
<td>catabfN</td>
<td>csinhfN</td>
<td>cimagfN</td>
</tr>
<tr>
<td>ccosfN</td>
<td>ctanhfN</td>
<td>CMPLXFN</td>
</tr>
<tr>
<td>csinfN</td>
<td>cexpfN</td>
<td>conjfN</td>
</tr>
<tr>
<td>cttanfN</td>
<td>clgofN</td>
<td>cprojfN</td>
</tr>
<tr>
<td>cacosshfN</td>
<td>cabsfN</td>
<td>crealfN</td>
</tr>
<tr>
<td>casinhfN</td>
<td>cpowfN</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Identifier</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfNx</td>
<td>catanhfNx</td>
<td>csqrtfNx</td>
</tr>
<tr>
<td>casinfNx</td>
<td>ccoshfNx</td>
<td>cargfNx</td>
</tr>
<tr>
<td>catabfNx</td>
<td>csinhfNx</td>
<td>cimagfNx</td>
</tr>
<tr>
<td>ccosfNx</td>
<td>ctanhfNx</td>
<td>CMPLXFNx</td>
</tr>
<tr>
<td>csinfNx</td>
<td>cexpfNx</td>
<td>conjfNx</td>
</tr>
<tr>
<td>cttanfNx</td>
<td>clgofNx</td>
<td>cprojfNx</td>
</tr>
</tbody>
</table>

© ISO/IEC 2018 – All rights reserved
X.8.3 `<math.h>`

[1] The following identifiers are defined or declared only if __STDC_WANT_IEC_60559_TYPES_EXT__ is defined as a macro at the point in the source file where `<math.h>` is first included:

```c
long_double_t
```

for supported types __FloatN:

<table>
<thead>
<tr>
<th>10</th>
<th>_FloatN_t</th>
<th>log1pfN</th>
<th>fromfpfN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HUGE_VAL_FN</td>
<td>log2fN</td>
<td>ufromfpfN</td>
</tr>
<tr>
<td></td>
<td>SNANF</td>
<td>logbfN</td>
<td>fromfpfxN</td>
</tr>
<tr>
<td></td>
<td>FP_FAST_FMAFN</td>
<td>modffN</td>
<td>ufromfpfxN</td>
</tr>
<tr>
<td></td>
<td>acosfN</td>
<td>scalbnfN</td>
<td>fmodfN</td>
</tr>
<tr>
<td></td>
<td>asinfN</td>
<td>scalblnfN</td>
<td>remainderfN</td>
</tr>
<tr>
<td></td>
<td>atanfN</td>
<td>cbrtfN</td>
<td>remquofN</td>
</tr>
<tr>
<td></td>
<td>atan2fN</td>
<td>fabsfN</td>
<td>copysignfN</td>
</tr>
<tr>
<td></td>
<td>cosfN</td>
<td>hypotfN</td>
<td>nanfN</td>
</tr>
<tr>
<td></td>
<td>sinfN</td>
<td>powfN</td>
<td>nextafterfN</td>
</tr>
<tr>
<td></td>
<td>tanfN</td>
<td>sqrtfN</td>
<td>nextupfN</td>
</tr>
<tr>
<td></td>
<td>acoshfN</td>
<td>erffN</td>
<td>nextdownfN</td>
</tr>
<tr>
<td></td>
<td>asinhfN</td>
<td>erfcfN</td>
<td>canonicalizefN</td>
</tr>
<tr>
<td></td>
<td>atanhfN</td>
<td>lgammafN</td>
<td>encodefN</td>
</tr>
<tr>
<td></td>
<td>coshfN</td>
<td>tgammafN</td>
<td>decodefN</td>
</tr>
<tr>
<td></td>
<td>sinhfN</td>
<td>ceilfN</td>
<td>fdimfN</td>
</tr>
<tr>
<td></td>
<td>tanhfN</td>
<td>floorfN</td>
<td>fmaxfN</td>
</tr>
<tr>
<td></td>
<td>expfN</td>
<td>nearbyintfN</td>
<td>fminfN</td>
</tr>
<tr>
<td></td>
<td>exp2fN</td>
<td>rintfN</td>
<td>fmaxmagfN</td>
</tr>
<tr>
<td></td>
<td>expm1fN</td>
<td>lrintfN</td>
<td>fminmagfN</td>
</tr>
<tr>
<td></td>
<td>frexpfN</td>
<td>llrintfN</td>
<td>fmafN</td>
</tr>
<tr>
<td></td>
<td>ilogbfN</td>
<td>roundfN</td>
<td>totalorderfN</td>
</tr>
<tr>
<td></td>
<td>ldexpfN</td>
<td>lroundfN</td>
<td>totalordermagfN</td>
</tr>
<tr>
<td></td>
<td>llogbfN</td>
<td>lroundfN</td>
<td>getpayloadfN</td>
</tr>
<tr>
<td></td>
<td>logfN</td>
<td>truncfN</td>
<td>setpayloadfN</td>
</tr>
<tr>
<td></td>
<td>log10fN</td>
<td>roundevenfN</td>
<td>setpayloadsigfN</td>
</tr>
</tbody>
</table>

for supported types __FloatNx:

```c
```
for supported types `_FloatM` and `_FloatN` where $M < N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FP_FAST_FMADDFN</code></td>
<td><code>FP_FAST_FMFMAFN</code></td>
</tr>
<tr>
<td><code>fMmulfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMSUBFN</code></td>
<td><code>FP_FAST_FMSQRTFN</code></td>
</tr>
<tr>
<td><code>fMdivfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMMULFN</code></td>
<td><code>fMaddfN</code></td>
</tr>
<tr>
<td><code>fMfmafN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMDIVFN</code></td>
<td><code>fMsubfN</code></td>
</tr>
<tr>
<td><code>fMsqrfN</code></td>
<td></td>
</tr>
</tbody>
</table>

for supported types `_FloatM` and `_FloatN` where $M \leq N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FP_FAST_FMADDFN</code></td>
<td><code>FP_FAST_FMFMAFN</code></td>
</tr>
<tr>
<td><code>fMmulfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMSUBFN</code></td>
<td><code>FP_FAST_FMSQRTFN</code></td>
</tr>
<tr>
<td><code>fMdivfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMMULFN</code></td>
<td><code>fMaddfN</code></td>
</tr>
<tr>
<td><code>fMfmafN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMDIVFN</code></td>
<td><code>fMsubfN</code></td>
</tr>
<tr>
<td><code>fMsqrfN</code></td>
<td></td>
</tr>
</tbody>
</table>

for supported types `_FloatMx` and `_FloatN` where $M < N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FP_FAST_FMXADDFN</code></td>
<td><code>FP_FAST_FMXFMAFN</code></td>
</tr>
<tr>
<td><code>fMmulfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMXSUBFN</code></td>
<td><code>FP_FAST_FMXSQRTFN</code></td>
</tr>
<tr>
<td><code>fMdivfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMXMULFN</code></td>
<td><code>fMaddfN</code></td>
</tr>
<tr>
<td><code>fMfmafN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMXDIVFN</code></td>
<td><code>fMsubfN</code></td>
</tr>
<tr>
<td><code>fMsqrfN</code></td>
<td></td>
</tr>
</tbody>
</table>

for supported types `_FloatMx` and `_FloatN` where $M < N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FP_FAST_FMXADDFN</code></td>
<td><code>FP_FAST_FMXFMAFN</code></td>
</tr>
<tr>
<td><code>fMmulfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMXSUBFN</code></td>
<td><code>FP_FAST_FMXSQRTFN</code></td>
</tr>
<tr>
<td><code>fMdivfN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMXMULFN</code></td>
<td><code>fMaddfN</code></td>
</tr>
<tr>
<td><code>fMfmafN</code></td>
<td></td>
</tr>
<tr>
<td><code>FP_FAST_FMXDIVFN</code></td>
<td><code>fMsubfN</code></td>
</tr>
<tr>
<td><code>fMsqrfN</code></td>
<td></td>
</tr>
</tbody>
</table>

for supported IEC 60559 arithmetic or non-arithmetic binary interchange formats of widths $M$ and $N$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fMencfN</code></td>
<td></td>
</tr>
</tbody>
</table>

for supported types `_DecimalN`, where $N \neq 32, 64, \text{and } 128$:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_DecimalN_t</code></td>
<td><code>logbdN</code></td>
</tr>
<tr>
<td><code>HUGE_VAL_DN</code></td>
<td><code>modfdN</code></td>
</tr>
<tr>
<td><code>SNANDN</code></td>
<td><code>scalbndN</code></td>
</tr>
<tr>
<td><code>FP_FAST_FMADN</code></td>
<td><code>scalblndN</code></td>
</tr>
<tr>
<td><code>acosdN</code></td>
<td><code>cbrtdN</code></td>
</tr>
<tr>
<td><code>asindN</code></td>
<td><code>fabsdN</code></td>
</tr>
<tr>
<td>Function</td>
<td>Function</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>atan2dN</td>
<td>hypotdN</td>
</tr>
<tr>
<td>cosdN</td>
<td>powdN</td>
</tr>
<tr>
<td>sindN</td>
<td>sqrtdN</td>
</tr>
<tr>
<td>tandN</td>
<td>erfN</td>
</tr>
<tr>
<td>acoshdN</td>
<td>lgammadN</td>
</tr>
<tr>
<td>asinhdN</td>
<td>tgammaN</td>
</tr>
<tr>
<td>atanhdN</td>
<td>ceildN</td>
</tr>
<tr>
<td>coshdN</td>
<td>floordN</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sinhN</td>
<td>nearbyintdN</td>
</tr>
<tr>
<td>tanhdN</td>
<td>rintdN</td>
</tr>
<tr>
<td>expdN</td>
<td>lrintdN</td>
</tr>
<tr>
<td>exp2dN</td>
<td>llrintdN</td>
</tr>
<tr>
<td>expmulN</td>
<td>rounddN</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>frexpN</td>
<td>lrounddN</td>
</tr>
<tr>
<td>ilogbdN</td>
<td>lrounddN</td>
</tr>
<tr>
<td>llogbdN</td>
<td>truncdN</td>
</tr>
<tr>
<td>ldexpN</td>
<td>roundevendN</td>
</tr>
<tr>
<td>logdN</td>
<td>fromfpdN</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>log10dN</td>
<td>uffromfpdN</td>
</tr>
<tr>
<td>log1pdN</td>
<td>fromfpxdN</td>
</tr>
<tr>
<td>log2dN</td>
<td>uffromfpxdN</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
| for supported types _DecimalN_x:

- **HUGE_VAL_D_Nx**: log2dN_x
- **SNAND_Nx**: logbdN_x
- **FP_FAST_FMA_D_Nx**: modfdN_x
- **acosdN_x**: scalbndN_x
- **asindN_x**: scalblndN_x
- **atan2dN_x**: cbrtdN_x
- **atan2dN_x**: nandN_x
- **cosdN_x**: hypotdN_x
- **sindN_x**: powdN_x
- **tandN_x**: sqrdN_x
- **tandN_x**: nextrdN_x
- **acosdN_x**: erfN_x
- **asinhN_x**: erfcN_x
- **atanhdN_x**: lgammadN_x
- **coshdN_x**: tgammaN_x
- **sinhdN_x**: ceildN_x
- **tanhdN_x**: floordN_x
- **5**:
- **expdN_x**: nearbyintdN_x
- **exp2dN_x**: rintdN_x
- **expmulN_x**: lrintdN_x
- **frexpN_x**: llrintdN_x
- **ilogbdN_x**: rounddN_x
- **llogbdN_x**: lrounddN_x
- **ldexpN_x**: llrounddN_x
- **logdN_x**: truncdN_x
- **log10dN_x**: roundevendN_x
- **log1pdN_x**: fromfpdN_x
- **log2dN_x**: uffromfpxdN_x
- **10**:                  |
| **25**            |                  |                  |
| **30**            |                  |                  |
| **35**            |                  |                  |
| **40**            |                  |                  |
| **45**            |                  |                  |

© ISO/IEC 2018 – All rights reserved
for supported types \texttt{\_Decimal} M and \texttt{\_Decimal} N where \( M < N \) and \( M \) and \( N \) are not both one of 32, 64, and 128:

\begin{Verbatim}
\texttt{FP\_FAST\_D\_MADD\_N} \quad \texttt{FP\_FAST\_D\_FMAD\_N} \quad \texttt{dMmuld\_N} \\
\texttt{FP\_FAST\_D\_MSUB\_N} \quad \texttt{FP\_FAST\_D\_MSQR\_TD\_N} \quad \texttt{dMdivd\_N} \\
\texttt{FP\_FAST\_D\_MUL\_N} \quad \texttt{dMadd\_N} \quad \texttt{dMfmad\_N} \\
\texttt{FP\_FAST\_D\_DIV\_N} \quad \texttt{dMsubd\_N} \quad \texttt{dMsqrt\_N}
\end{Verbatim}

for supported types \texttt{\_Decimal} M and \texttt{\_Decimal} N x where \( M \leq N \):

\begin{Verbatim}
\texttt{FP\_FAST\_D\_MADD\_N\_X} \quad \texttt{FP\_FAST\_D\_FMAD\_N\_X} \quad \texttt{dMmuld\_N\_X} \\
\texttt{FP\_FAST\_D\_MSUB\_N\_X} \quad \texttt{FP\_FAST\_D\_MSQR\_TD\_N\_X} \quad \texttt{dMdivd\_N\_X} \\
\texttt{FP\_FAST\_D\_MUL\_N\_X} \quad \texttt{dMadd\_N\_X} \quad \texttt{dMfmad\_N\_X} \\
\texttt{FP\_FAST\_D\_DIV\_N\_X} \quad \texttt{dMsubd\_N\_X} \quad \texttt{dMsqrt\_N\_X}
\end{Verbatim}

for supported types \texttt{\_Decimal} M x and \texttt{\_Decimal} N where \( M < N \):

\begin{Verbatim}
\texttt{FP\_FAST\_D\_MXADD\_N\_X} \quad \texttt{FP\_FAST\_D\_MXFMAD\_N\_X} \quad \texttt{dMxmulp\_N\_X} \\
\texttt{FP\_FAST\_D\_MSUB\_N\_X} \quad \texttt{FP\_FAST\_D\_MXXQR\_TD\_N\_X} \quad \texttt{dMdivp\_N\_X} \\
\texttt{FP\_FAST\_D\_MUL\_N\_X} \quad \texttt{dMxadd\_N\_X} \quad \texttt{dMxmpd\_N\_X} \\
\texttt{FP\_FAST\_D\_DIV\_N\_X} \quad \texttt{dMxsubd\_N\_X} \quad \texttt{dMxsqrt\_N\_X}
\end{Verbatim}

for supported types \texttt{\_Decimal} M x and \texttt{\_Decimal} N x where \( M < N \):

\begin{Verbatim}
\texttt{FP\_FAST\_D\_MXADD\_N\_X\_X} \quad \texttt{FP\_FAST\_D\_MXFMAD\_N\_X\_X} \quad \texttt{dMxmulp\_N\_X\_X} \\
\texttt{FP\_FAST\_D\_MSUB\_N\_X\_X} \quad \texttt{FP\_FAST\_D\_MXXQR\_TD\_N\_X\_X} \quad \texttt{dMdivp\_N\_X\_X} \\
\texttt{FP\_FAST\_D\_MUL\_N\_X\_X} \quad \texttt{dMxadd\_N\_X\_X} \quad \texttt{dMxmpd\_N\_X\_X} \\
\texttt{FP\_FAST\_D\_DIV\_N\_X\_X} \quad \texttt{dMxsubd\_N\_X\_X} \quad \texttt{dMxsqrt\_N\_X\_X}
\end{Verbatim}

for supported IEC 60559 arithmetic and non-arithmetic decimal interchange formats of widths \( M \) and \( N \):

\begin{Verbatim}
dMencdec\_d\_N \quad \texttt{dMencbind\_d\_N}
\end{Verbatim}

\section*{X.8.4 \texttt{\langle stdlib.h\rangle}}

[1] The following identifiers are declared only if \texttt{\_STDC\_WANT\_IEC\_60559\_TYPES\_EXT} is defined as a macro at the point in the source file where \texttt{\langle stdlib.h\rangle} is first included:

for supported types \texttt{\_Float} N:

\begin{Verbatim}
\texttt{strfromf\_N} \quad \texttt{strtof\_N}
\end{Verbatim}

for supported types \texttt{\_Float} N x:

\begin{Verbatim}
\texttt{strfromfx\_N} \quad \texttt{strtofx\_N}
\end{Verbatim}

for supported types \texttt{\_Decimal} N, where \( N \neq 32, 64, \) and \( 128 \):

\begin{Verbatim}
\texttt{strfromd\_N} \quad \texttt{strtod\_N}
\end{Verbatim}

for supported types \texttt{\_Decimal} N x:

\begin{Verbatim}
\texttt{strfromdx\_N} \quad \texttt{strtodx\_N}
\end{Verbatim}
for supported IEC 60559 arithmetic and non-arithmetic binary interchange formats of width $N$:

```c
strfromencfN
strtoencfN
```

for supported IEC 60559 arithmetic and non-arithmetic decimal interchange formats of width $N$:

```c
strfromencdecN
strtoencdecN
strfromencbindN
strtoencbindN
```

### 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. Conversely, for each such type that the implementation does not provide, `<complex.h>` shall not declare the associated functions.

#### 7.3.5 Trigonometric functions

```c
_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

```c
_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);
```
7.3.7 Exponential and logarithmic functions

_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);

7.3.8 Power and absolute value functions

_FloatN complex cabsfN(_FloatN complex z);
_FloatNx complex 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 7.31.1 (Future library directions for <complex.h>), 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 for standard floating types to also apply to binary floating types.

[2] The rounding control pragma (7.6.2)

```
#pragma STDC FENV_ROUND direction
```

affects operations for standard and binary floating types. direction shall be one of the names of the supported rounding direction macros for use with fegetround and fesetround (7.6), or FE_DYNAMIC.

[3] In 7.6.2, the table entitled

Functions affected by constant rounding modes – for standard floating types

applies to binary floating types, as well as standard 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 acos, acosl, acosf, and acosfx as well as acos). The fMencfN, strfromencfN, and strtoencfN functions for binary interchange types are also affected by constant rounding modes.

[4] 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 acosdx). The dMencbindN, dMencdecN, strfromencbindN, strfromencdecN, strtoencbindN, and strtoencdecN functions for decimal interchange types are also affected by constant rounding modes.

[5] The fegetround function (7.6.5.2) gets the current value of the dynamic rounding direction mode for operations for standard and binary floating types. The fesetround function (7.6.5.5) sets the dynamic rounding direction mode to the rounding direction represented by its argument round for operations for standard and binary floating types.

X.11 Mathematics <math.h>

[1] This subclause specifies 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 macros and declare the associated functions. Conversely, for each such type that the implementation does not provide, <math.h> shall not define the associated macros or declare the associated functions unless explicitly specified otherwise.
[6] With the types

\[
\begin{align*}
&\text{float}_t \\
&\text{double}_t
\end{align*}
\]

in 7.12 are included the type

\[
\text{long\_double}_t
\]

and for each supported type \_Float\textit{N}, the type

\[
\_\text{Float}\textit{N}_t
\]

and for each supported type \_Decimal\textit{N}, the type

\[
\_\text{Decimal}\textit{N}_t
\]

These are floating types, such that:

- each of the types has at least the range and precision of the corresponding real floating type;
- \text{long\_double}_t has at least the range and precision of \text{double}_t;
- \_Float\textit{N}_t has at least the range and precision of \_Float\textit{M}_t if \textit{N} > \textit{M};
- \_Decimal\textit{N}_t has at least the range and precision of \_Decimal\textit{M}_t if \textit{N} > \textit{M}.

If \texttt{FLT\_RADIX} is 2 and \texttt{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 \texttt{FLT\_EVAL\_METHOD} (X.3) to be used for evaluating operations and constants of that standard or binary floating type. If \texttt{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 \texttt{DEC\_EVAL\_METHOD} (X.3) to be used for evaluating operations and constants of that decimal floating type.

[Jens, I’m preparing an example to go here.]

X.11.1 Macros

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

[2] The macros

\[
\begin{align*}
&\text{HUGE\_VAL\_F}\textit{N} \\
&\text{HUGE\_VAL\_D}\textit{N} \\
&\text{HUGE\_VAL\_F}\textit{NX} \\
&\text{HUGE\_VAL\_D}\textit{NX}
\end{align*}
\]

expand to constant expressions of types \_Float\textit{N}, \_Decimal\textit{N}, \_Float\textit{Nx}, and \_Decimal\textit{Nx}, respectively, representing positive infinity.

[3] The signaling NaN macros

\[
\begin{align*}
&\text{SNANF}\textit{N} \\
&\text{SNAND}\textit{N} \\
&\text{SNANF}\textit{NX} \\
&\text{SNAND}\textit{NX}
\end{align*}
\]
expand to constant expressions of types _FloatN, _DecimalN, _FloatNx, and _DecimalNx, 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.

[4] The macros

\[
\begin{align*}
FP\_FAST\_FMADF\_N \\
FP\_FAST\_FMADD\_N \\
FP\_FAST\_FMADFF\_N \\
FP\_FAST\_FMADD\_F \\
FP\_FAST\_FMADD\_D \\
FP\_FAST\_FMADD\_M \\
FP\_FAST\_FMADD\_X \\
FP\_FAST\_FMADD\_F \_X \\
FP\_FAST\_FMADD\_D \_X \\
\end{align*}
\]

are, respectively, _FloatN, _DecimalN, _FloatNx, and _DecimalNx 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_DAADDL, etc.

[6] For \( M < N \), the macros

\[
\begin{align*}
FP\_FAST\_FMADD\_F \_D \\
FP\_FAST\_FMADD\_D \_F \\
FP\_FAST\_FMADD\_D \_D \\
FP\_FAST\_FMADD\_D \_M \\
FP\_FAST\_FMADD\_D \_X \\
FP\_FAST\_FMADD\_F \_D \_X \\
FP\_FAST\_FMADD\_D \_F \_X \\
FP\_FAST\_FMADD\_D \_D \_X \\
FP\_FAST\_FMADD\_D \_M \_X \\
FP\_FAST\_FMADD\_D \_X \_X \\
\end{align*}
\]

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 \( M \leq N \), the macros

\[
\begin{align*}
FP\_FAST\_FMADD\_F \_D \_X \\
FP\_FAST\_FMADD\_D \_F \_X \\
FP\_FAST\_FMADD\_D \_D \_X \\
FP\_FAST\_FMADD\_D \_M \_X \\
FP\_FAST\_FMADD\_D \_X \_X \\
FP\_FAST\_FMADD\_F \_D \_X \_X \\
FP\_FAST\_FMADD\_D \_F \_X \_X \\
FP\_FAST\_FMADD\_D \_D \_X \_X \\
FP\_FAST\_FMADD\_D \_M \_X \_X \\
FP\_FAST\_FMADD\_D \_X \_X \_X \\
\end{align*}
\]

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

\[
\begin{align*}
\text{FP\_FAST\_FMXADDFN} \\
\text{FP\_FAST\_FMXSUBFN} \\
\text{FP\_FAST\_FMXMULFN} \\
\text{FP\_FAST\_FMXDIVFN} \\
\text{FP\_FAST\_FMXFMAFN} \\
\text{FP\_FAST\_FMXSQRTFN} \\
\text{FP\_FAST\_DMXADDDN} \\
\text{FP\_FAST\_DMXSUBDN} \\
\text{FP\_FAST\_DMXMULDN} \\
\text{FP\_FAST\_DMXDIVDN} \\
\text{FP\_FAST\_DMXFADN} \\
\text{FP\_FAST\_DMXSQRTDN}
\end{align*}
\]

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

\[
\begin{align*}
\text{FP\_FAST\_FMXADDFNX} \\
\text{FP\_FAST\_FMXSUBFNX} \\
\text{FP\_FAST\_FMXMULFNX} \\
\text{FP\_FAST\_FMXDIVFNX} \\
\text{FP\_FAST\_FMXFMAFNX} \\
\text{FP\_FAST\_FMXSQRTFNX} \\
\text{FP\_FAST\_DMXADDDNX} \\
\text{FP\_FAST\_DMXSUBDNX} \\
\text{FP\_FAST\_DMXMULDNX} \\
\text{FP\_FAST\_DMXDIVDNX} \\
\text{FP\_FAST\_DMXFADNX} \\
\text{FP\_FAST\_DMXSQRTDNX}
\end{align*}
\]

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

\[
\begin{align*}
\text{\_FloatN \ acosfN(_\text{FloatN } x);} \\
\text{\_FloatNx \ acosfNx(_\text{FloatNx } x);} \\
\text{\_DecimalN \ acosdN(_\text{DecimalN } x);} \\
\text{\_DecimalNx \ acosdNx(_\text{DecimalNx } x);} \\
\text{\_FloatN \ asinfN(_\text{FloatN } x);} \\
\text{\_FloatNx \ asinfNx(_\text{FloatNx } x);} \\
\text{\_DecimalN \ asindN(_\text{DecimalN } x);} \\
\text{\_DecimalNx \ asindNx(_\text{DecimalNx } x);} \\
\end{align*}
\]
_FloatN atanf(_FloatN x);
_FloatN atanfNx(_FloatN x);
DecimalN atandN(_DecimalN x);
DecimalN atandNx(_DecimalN x);

_FloatN atan2f(_FloatN y, _FloatN x);
_FloatN atan2fNx(_FloatN y, _FloatN x);
DecimalN atan2d(_DecimalN y, _DecimalN x);
DecimalN atan2dNx(_DecimalN y, _DecimalN x);

_FloatN cosf(_FloatN x);
_FloatN cosfNx(_FloatN x);
DecimalN cosd(_DecimalN x);
DecimalN cosdNx(_DecimalN x);

_FloatN sinf(_FloatN x);
_FloatN sinfNx(_FloatN x);
DecimalN sind(_DecimalN x);
DecimalN sindNx(_DecimalN x);

_FloatN tanf(_FloatN x);
_FloatN tanfNx(_FloatN x);
DecimalN tand(_DecimalN x);
DecimalN tandNx(_DecimalN x);

7.12.5 Hyperbolic functions

_FloatN acoshf(_FloatN x);
_FloatN acoshfNx(_FloatN x);
DecimalN acoshd(_DecimalN x);
DecimalN acoshdNx(_DecimalN x);

_FloatN asinhf(_FloatN x);
_FloatN asinhfNx(_FloatN x);
DecimalN asinhd(_DecimalN x);
DecimalN asinhdNx(_DecimalN x);

_FloatN atanhf(_FloatN x);
_FloatN atanhfNx(_FloatN x);
DecimalN atanhd(_DecimalN x);
DecimalN atanhdNx(_DecimalN x);

_FloatN coshf(_FloatN x);
_FloatN coshfNx(_FloatN x);
DecimalN coshd(_DecimalN x);
DecimalN coshdNx(_DecimalN x);

_FloatN sinhf(_FloatN x);
_FloatN sinhfNx(_FloatN x);
DecimalN sinhN(_DecimalN x);
DecimalN sinhNx(_DecimalN x);
7.12.6 Exponential and logarithmic functions

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

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

_FloatN exp2fN(_FloatN x);
_FloatNx exp2fNx(_FloatNx x);
DecimalN exp2dN(_DecimalN x);
DecimalNx exp2dNx(_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 frexpdxN(_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);
7.12.7 Power and absolute-value functions

_FloatN log1pfN(_FloatN x);
_FloatN log1pfNx(_FloatN x);
_DecimalN log1pdN(_DecimalN x);
_DecimalN log1pdNx(_DecimalN x);

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

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

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

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

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

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

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

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

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

_FloatN sqrtfN(_FloatN x);
_FloatN sqrtfNx(_FloatN x);
7.12.8 Error and gamma functions

_preDecimalN sqrt(N(_Decimal N x));
_preDecimalNx sqrt(Nx(_Decimal Nx x));

_FloatN erf(N(_Float N x));
_FloatNx erf Nx(_Float Nx x);
_PreDecimalN erfd(N(_Decimal N x));
_PreDecimalNx erfdNx(_Decimal Nx x);

_FloatN erf(Nc fN(_Float N x));
_Float Nx erf c f Nx(_Float N x);
_PreDecimalN erfc d(N(_Decimal N x));
_PreDecimalNx erfc dNx(_Decimal Nx x);

_FloatN lg(mafN(_Float N x));
_FloatNx lg maf Nx(_Float Nx x);
_PreDecimalN lg m a d N(_Decimal N x);
_PreDecimalNx lg m a d Nx(_Decimal Nx x);

_FloatN tgammaf(_Float N x));
_Float Nx tgammaf Nx(_Float Nx x);
_PreDecimalN tgamma d N(_Decimal N x);
_PreDecimalNx tgamma d Nx(_Decimal Nx x);

7.12.9 Nearest integer functions

_FloatN ceilf(_Float N x));
_FloatNx ceilf Nx(_Float Nx x);
_PreDecimalN ceild(N(_Decimal N x));
_PreDecimalNx ceildNx(_Decimal Nx x);

_FloatN floorf(_Float N x));
_FloatNx floorf Nx(_Float Nx x);
_PreDecimalN floord(N(_Decimal N x));
_PreDecimalNx floordNx(_Decimal Nx x);

_FloatN nearbyintf(_Float N x));
_FloatNx nearbyintf Nx(_Float Nx x);
_PreDecimalN nearbyintd(N(_Decimal N x));
_PreDecimalNx nearbyintdNx(_Decimal Nx x);

_FloatN rintf(_Float N x));
_FloatNx rintf Nx(_Float Nx x);
_PreDecimalN rintl N(_Decimal N x));
_PreDecimalNx rintd Nx(_Decimal Nx x);

long int lrintf(_Float N x));
long int lrintf Nx(_Float Nx x);
long int lrintd(N(_Decimal N x));
long int lrintdNx(_Decimal Nx x);
long long int llrintfN(_FloatN x);
long long int llrintfx(_FloatNx x);
long long int llrintdN(_DecimalN x);
long long int llrintdx(_DecimalNx x);

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

long int lroundfN(_FloatN x);
long int lroundfx(_FloatNx x);
lrounddN(_DecimalN x);
lrounddx(_DecimalNx x);

_FloatN roundevenfN(_FloatN x);
_FloatNx roundevenfx(_FloatNx x);
_DecimalN roundevenfN(_DecimalN x);
_DecimalN roundevenfx(_DecimalNx x);

_FloatN truncfN(_FloatN x);
_FloatNx truncfx(_FloatNx x);
_DecimalN truncdN(_DecimalN x);
_DecimalN truncdx(_DecimalNx x);

intmax_t fromfpfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpfxN(_FloatNx x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpdxN(_DecimalNx x, int round, unsigned int width);
uintmax_t ufromfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfxN(_FloatNx x, int round, unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpdxN(_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);

_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 quantumN(_DecimalN x);
_DecimalNx quantumNx(_DecimalNx x);
long long int llquantexpdN(_DecimalN x);
long long int llquantexpNx(_DecimalNx x);

7.12.12 Maximum, minimum, and positive difference functions

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

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

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

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

_FloatN fminmagf(_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 fnafN(_FloatN x, _FloatN y, _FloatN z);  
_FloatNx fnafNx(_FloatNx x, _FloatNx y, _FloatNx z);
_DecimalN fnadN(_DecimalN x, _DecimalN y, _DecimalN z);
_DecimalNx fnadNx(_DecimalNx x, _DecimalNx y, _DecimalNx z);

7.12.14 Functions that round result to narrower type

_FloatM fmadNfN(_FloatN x, _FloatN y);  // M < N
_FloatM fmadNfNx(_FloatNx x, _FloatNx y);  // M <= N
_FloatM fmadfN(_FloatN x, _FloatN y);  // M < N
_FloatM fmadfNx(_FloatNx x, _FloatNx y);  // M < N
_DecimalM dMaddD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMaddDx(_DecimalNx x, _DecimalN y); // M <= N
_decimalM dMxaddD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMxaddDx(_DecimalNx x, _DecimalN y); // M < N

_FloatM fMsubF(_FloatN x, _FloatN y); // M < N
_FloatM fMsubFx(_FloatNx x, _FloatN y); // M <= N
_FloatM fMsubFxN(_FloatN x, _FloatN y); // M < N
_FloatM fMexsubF(_FloatN x, _FloatN y); // M < N
_decimalM dMsubD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMsubDx(_DecimalNx x, _DecimalN y); // M <= N
_decimalM dMxsubD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMxsubDx(_DecimalNx x, _DecimalN y); // M < N

_FloatM fMmulF(_FloatN x, _FloatN y); // M < N
_FloatM fMmulFx(_FloatNx x, _FloatN y); // M <= N
_FloatM fMmulFxN(_FloatN x, _FloatN y); // M < N
_FloatM fMxmulF(_FloatN x, _FloatN y); // M < N
_decimalM dMmulD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMmulDx(_DecimalNx x, _DecimalN y); // M <= N
_decimalM dMxmulD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMxmulDx(_DecimalNx x, _DecimalN y); // M < N

_FloatM fMdivF(_FloatN x, _FloatN y); // M < N
_FloatM fMdivFx(_FloatNx x, _FloatN y); // M <= N
_FloatM fMdivFxN(_FloatN x, _FloatN y); // M < N
_FloatM fMxdivF(_FloatN x, _FloatN y); // M < N
_decimalM dMdivD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMdivDx(_DecimalNx x, _DecimalN y); // M <= N
_decimalM dMxdivD(_DecimalN x, _DecimalN y); // M < N
_decimalM dMxdivDx(_DecimalNx x, _DecimalN y); // M < N

_FloatM fMfmaF(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMfmaFx(_FloatNx x, _FloatN y, _FloatN z); // M <= N
_FloatM fMxmaF(_FloatN x, _FloatN y, _FloatN z); // M < N
_decimalM dMfmadD(_DecimalN x, _DecimalN y, _DecimalN z); // M < N
_decimalM dMfmadDx(_DecimalNx x, _DecimalN y, _DecimalN z); // M <= N
_decimalM dMxfmadD(_DecimalN x, _DecimalN y, _DecimalN z); // M < N
_decimalM dMxfmadDx(_DecimalNx x, _DecimalN y, _DecimalN z); // M < N

_FloatM fMsqrtF(_FloatN x); // M < N
_FloatM fMsqrtFx(_FloatNx x); // M <= N
_FloatM fMsqrtFxN(_FloatN x); // M < N
_FloatM fMxsqrtF(_FloatN x); // M < N
_FloatM fMxsqrtFx(_FloatNx x); // M < N
_FloatM fMxsqrtFxN(_FloatN 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

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 _FloatFx *x, const _FloatFx *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(_FloatFx *res, _FloatFx pl);
int setpayloadsigdN(_DecimalN *res, _DecimalN pl);
int setpayloadsigdNx(_DecimalNx *res, _DecimalNx pl);

[2] The specification of the freq functions (7.12.6.7) applies 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 \( x \times 2^{\text{exp}} \).

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

[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.

[2] Non-arithmetic interchange formats are not associated with floating types. Arrays of element type unsigned char are used as parameters for conversions functions, to represent encodings in interchange formats that might be non-arithmetic formats.
X.11.3.1 Encode and decode functions

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

```
#include <math.h>
void encodefN(unsigned char * restrict encptr,
              const_FloatN * restrict xptr);
```

Description

The encodefN functions convert *xptr into an IEC 60559 binaryN 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

The encodefN functions return no value.

X.11.3.1.2 The decodefN functions

Synopsis

```
#include <math.h>
void decodefN (_FloatN * restrict xptr,
               const unsigned char * restrict encptr);
```

Description

The decodefN functions interpret the N/8 element array pointed to by encptr as an IEC 60559 binaryN 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

The decodefN functions return no value.

X.11.3.2 Encoding-to-encoding conversion functions

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. An implementation shall provide both dMencodedN and dMenckbind functions for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.
X.11.3.2.1 The $f_{MencfN}$ functions

Synopsis

[1]  #include <math.h>
    void $f_{MencfN}$(unsigned char * restrict enc$M$ptr,
                   const unsigned char * restrict enc$N$ptr);

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.

X.11.3.2.2 The $d_{Mencdec}d_{N}$ and $d_{Mencbind}N$ functions

Synopsis

[1]  #include <math.h>
    void $d_{Mencdec}d_{N}$(unsigned char * restrict enc$M$ptr,
                          const unsigned char * restrict enc$N$ptr);
    void $d_{Mencbind}N$(unsigned char * restrict enc$M$ptr,
                        const unsigned char * restrict enc$N$ptr);

Description

[2] These functions convert between IEC 60559 decimal interchange formats that use the same encoding scheme. The $d_{Mencdec}d_{N}$ functions convert between formats using the encoding scheme based on decimal encoding of the significand. The $d_{Mencbind}N$ 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. 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. 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

\[
\text{int strfromfN(char * restrict s, size_t n, const char * restrict format, _FloatN fp);} \\
\text{int strfromfNx(char * restrict s, size_t n, const char * restrict format, _FloatNx fp);} \\
\]

It encompasses the prototypes in 7.22.1.4 by replacing them with

\[
\text{int strfromdN(char * restrict s, size_t n, const char * restrict format, _DecimalN fp);} \\
\text{int strfromdNx(char * restrict s, size_t n, const char * restrict format, _DecimalNx fp);} \\
\]

[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

\[
\text{_FloatN strtofN(const char * restrict nptr, char ** restrict endptr);} \\
\text{_FloatNx strtofNx(const char * restrict nptr, char ** restrict endptr);} \\
\]

It encompasses the prototypes in 7.22.1.6 by replacing them with

\[
\text{_DecimalN strtodN(const char * restrict nptr, char ** restrict endptr);} \\
\text{_DecimalNx strtodNx(const char * restrict nptr, char ** restrict endptr);} \\
\]

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

X.12.3 String from encoding

[1] An implementation shall declare the \text{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 \text{strfromencodedN} and \text{strfromencbindN} 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 \texttt{strfromencfN} functions

\textbf{Synopsis}

[1] \begin{verbatim}
#include <stdlib.h>
int strfromencfN(char * restrict s, size_t n,
    const char * restrict format,
    const unsigned char * restrict encptr);
\end{verbatim}

\textbf{Description}

[2] The \texttt{strfromencfN} functions are similar to the \texttt{strfromfN} functions, except the input is the value of the $N/8$ element array pointed to by \texttt{encptr}, interpreted as an IEC 60559 binary encoding. The order of bytes in the arrays is implementation-defined.

\textbf{Returns}

[3] The \texttt{strfromencfN} functions return the same values as corresponding \texttt{strfromfN} functions.

X.12.3.2 The \texttt{strfromencdecN} and \texttt{strfromencbindN} functions

\textbf{Synopsis}

[1] \begin{verbatim}
#include <stdlib.h>
int strfromencdecN(char * restrict s, size_t n,
    const char * restrict format,
    const unsigned char * restrict encptr);
int strfromencbindNx(char * restrict s, size_t n,
    const char * restrict format,
    const unsigned char * restrict encptr);
\end{verbatim}

\textbf{Description}

[2] The \texttt{strfromencdecN} functions are similar to the \texttt{strfromdN} functions except the input is the value of the $N/8$ element array pointed to by \texttt{encptr}, interpreted as an IEC 60559 decimal encoding in the coding scheme based on decimal encoding of the significand. The \texttt{strfromencbindN} functions are similar to the \texttt{strfromdN} functions except the input is the value of the $N/8$ element array pointed to by \texttt{encptr}, interpreted as an IEC 60559 decimal encoding in the coding scheme based on binary encoding of the significand. The order of bytes in the arrays is implementation-defined.

\textbf{Returns}

[3] The \texttt{strfromencdecN} and \texttt{strfromencbindN} functions return the same values as corresponding \texttt{strfromdN} functions.

X.12.4 String to encoding

X.12.4.1 The \texttt{strtoencfN} functions

\textbf{Synopsis}

[1] \begin{verbatim}
#include <stdlib.h>
void strtoencfN(unsigned char * restrict encptr,
    const char * restrict nptr, char ** restrict endptr);
\end{verbatim}
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 `strtoencdecdN` and `strtoencbindN` functions

Synopsis

[1] 
```c
#include <stdlib.h>
void strtoencdecdN(unsigned char * restrict encptr, const char * restrict nptr, char ** restrict endptr);
void strtoencbindN(unsigned char * restrict encptr, const char * restrict nptr, char ** restrict endptr);
```

Description

[2] The `strtoencdecdN` 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 `strtoencdecdN` 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 floating type of radix 2 and another argument is of decimal floating type, the behavior is undefined.

[3] Use of the macro `carg, cimag, conj, cproj, or creal` with any argument of standard floating type, floating type of radix 2, or complex type, invokes a complex function. Use of the macro with an argument of a decimal floating type results in undefined behavior.

[4] 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 `fM, fMx, dM, or dMx` prefix are:
All arguments are generic. If any argument is not real, use of the macro results in undefined behavior. If the macro prefix is \texttt{f} or \texttt{d}, use of an argument of interchange or extended floating type results in undefined behavior. If the macro prefix is \texttt{fM}, or \texttt{fMx}, use of an argument of standard or decimal floating type results in undefined behavior. If the macro prefix is \texttt{dM} or \texttt{dMx}, use of an argument of standard or binary floating type results in undefined behavior. The function invoked is determined as follows:

- Arguments that have integer type are regarded as having type \texttt{double} if the macro prefix is \texttt{f} or \texttt{d}, as having type \texttt{__Float64} if the macro prefix is \texttt{fM} or \texttt{fMx}, and as having type \texttt{__Decimal64} if the macro prefix is \texttt{dM} or \texttt{dMx}.

- If the function has exactly one generic parameter, the type determined is the type of the argument.

- If the function has exactly two generic parameters, the type determined is the type determined by the usual arithmetic conversions (X.4.2) applied to the arguments.

- If the function has three generic parameters, the type determined is the type determined by applying the usual arithmetic conversions (X.4.2) twice, first to the first two arguments, then to that result type and the third argument.

- If no function with the given prefix has the parameter type determined above, the parameter type is determined from the prefix as follows:

<table>
<thead>
<tr>
<th>\texttt{f}</th>
<th>\texttt{d}</th>
<th>\texttt{dM}</th>
<th>\texttt{dMx}</th>
<th>\texttt{f}</th>
<th>\texttt{d}</th>
<th>\texttt{dM}</th>
<th>\texttt{dMx}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{double}</td>
<td>\texttt{long double}</td>
<td>\texttt{__FloatN} for minimum \texttt{N &gt; M} if supported, else \texttt{__FloatMx}</td>
<td>\texttt{__DecimalN} for minimum \texttt{N &gt; M} if supported, else \texttt{__DecimalMx}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{fM}</td>
<td>\texttt{fMx}</td>
<td>\texttt{dM}</td>
<td>\texttt{dMx}</td>
<td>\texttt{fM}</td>
<td>\texttt{fMx}</td>
<td>\texttt{dM}</td>
<td>\texttt{dMx}</td>
</tr>
</tbody>
</table>

**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;
__Float128 f128;
__Float64x complex f64xc;
```
functions invoked by use of type-generic macros are shown in the following table:

<table>
<thead>
<tr>
<th>macro use</th>
<th>invokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cos(f64xc)</td>
<td>ccosf64x(f64xc)</td>
</tr>
<tr>
<td>pow(dc, f128)</td>
<td>cpowf128(dc, f128)</td>
</tr>
<tr>
<td>fmax(f64, d)</td>
<td>fmaxf64(f64, d)</td>
</tr>
<tr>
<td>fmax(d, f32x)</td>
<td>fmax(d, f32x), the function, if the set of values of _Float32x is a subset of (or equivalent to) the set of values of double, or fmaxf32x(d, f32x), if the set of values of double is a proper subset of the set of values of _Float32x, or undefined, if neither of the sets of values of double and _Float32x is a subset of the other (and the sets are not equivalent)</td>
</tr>
<tr>
<td>pow(f32x, n)</td>
<td>powf32x(f32x, n)</td>
</tr>
<tr>
<td>fsub(d, ld)</td>
<td>fsubl</td>
</tr>
<tr>
<td>f32add(f64x, f64)</td>
<td>f32addf64x</td>
</tr>
<tr>
<td>f32sqrt(n)</td>
<td>f32sqrtf64</td>
</tr>
<tr>
<td>f32mul(f128, f32x)</td>
<td>f32mulf128 if _Float128 is at least as wide as _Float32x, or f32mulf32x if _Float32x is wider than _Float128 f32fma(f32x, n, f32x)</td>
</tr>
<tr>
<td>ddiv(ld, f128)</td>
<td>undefined</td>
</tr>
<tr>
<td>f32fma(f64, d, f64)</td>
<td>undefined</td>
</tr>
<tr>
<td>fmul(dc, d)</td>
<td>f32addf32x(f32, f32)</td>
</tr>
<tr>
<td>f32sqrt(f32)</td>
<td>f32sqrtf64x</td>
</tr>
<tr>
<td>f64div(f32x, f32x)</td>
<td>f64divf128be(f32x, f32x)</td>
</tr>
</tbody>
</table>