Annex X

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 extension 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 C17:

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, 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>(emax), maximum exponent (e)</td>
<td>15</td>
<td>127</td>
<td>1023</td>
<td>16383</td>
<td>(2^{(N-p-1)} - 1)</td>
</tr>
</tbody>
</table>

#### Encoding parameters

<table>
<thead>
<tr>
<th></th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
<th>decimal(N(N \geq 32))</th>
</tr>
</thead>
<tbody>
<tr>
<td>bias, (E-e)</td>
<td>101</td>
<td>398</td>
<td>6176</td>
<td></td>
</tr>
<tr>
<td>(sign) bit</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(w), exponent 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>

The function `round()` in the table above rounds to the nearest integer. For example, binary256 would have \(p = 237\) and \(emax = 262143\).

### 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>(emax), 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></th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
<th>decimal(N(N \geq 32))</th>
</tr>
</thead>
<tbody>
<tr>
<td>bias, (E-e)</td>
<td>101</td>
<td>398</td>
<td>6176</td>
<td></td>
</tr>
<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 \(emax = 1572864\).

[2] Types designated

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

\_\_\_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_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>Extended format parameters for floating-point numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>(p) digits (\geq)</td>
</tr>
<tr>
<td>(emax) (\geq)</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 \_Decimal32\_ 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 include similar types whose 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 include similar types whose 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 5.2.4.2a of the three decimal floating types to include this annex’s decimal floating types. The prefix FLT indicates a binary interchange floating type or a non-arithmetic binary interchange format of width \(N\). The prefix FLT\(N\) indicates a binary extended floating type that extends a basic format of width \(N\). The prefix DEC\(N\) indicates a decimal extended floating type of width \(N\). The prefix DEC\(NX\) indicates a decimal extended floating type that extends a basic format of width \(N\).

The prefix FLT\(N\)\_indicates a binary interchange floating type or a non-arithmetic binary interchange 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, \(<\text{float.h}>\) 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 FLT\(N\)_DECIMAL_DIG and FLT\(N\)_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 \(<\text{math.h}>\) and the string-from-encoding (X.12.3) and string-to-encoding (X.12.4) functions in \(<\text{stdlib.h}>\).

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 interchange and extended floating types:

- 1 indeterminable;
- 0 evaluate all operations and constants, whose semantic type has at most the range and precision 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 has at most the range and precision 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 has at most the range and precision of long double, to the range and precision of long double; evaluate all other operations and constants to the range and precision of the semantic type;

\(N\), where \_Float\(N\) is a supported interchange floating type

evaluate operations and constants, whose semantic type has at most the range and precision of the \_Float\(N\) type, to the range and precision of the \_Float\(N\) type;

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

\(N + 1\), where \_Float\(Nx\) is a supported extended floating type

evaluate operations and constants, whose semantic type has at most the range and precision of the \_Float\(Nx\) type, to the range and precision of the \_Float\(Nx\) type;

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 interchange and extended floating types is implementation-defined.

[3] The implementation-defined value of the macro DEC\_EVAL\_METHOD (5.2.4.2.2a) characterizes the use of evaluation formats for decimal interchange and extended 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 has at most the range and precision of the \_Decimal64 type, to the range and precision of the \_Decimal64 type.

\(N\), where \_Decimal64\(N\) is a supported interchange floating type

evaluate operations and constants, whose semantic type has at most the range and precision of the \_Decimal64\(N\) type, to the range and precision of the \_Decimal64\(N\) type;

evaluate all other operations and constants to the range and precision of the semantic type.

\(N + 1\), where \_Decimal64\(Nx\) is a supported extended floating type

evaluate operations and constants, whose semantic type has at most the range and precision of the \_Decimal64\(Nx\) type, to the range and precision of the \_Decimal64\(Nx\) type;

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

5 \( N \), where _DecimalN 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;

10 \( N + 1 \), where _DecimalNx 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 for binary, 10 for decimal)

25 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, \( p \)

25 

25 FLT_N_MANT_DIG

25 FLT_NX_MANT_DIG

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

30 

30 DEC_N_MANT_DIG

30 DEC_NX_MANT_DIG

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

35 

35 FLT_N_DECIMAL_DIG

35 FLT_NX_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, \( \left[ ( p - 1) \log_{10} 2 \right] \)

40 

40 FLT_N_DIG

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

\[
\text{FLT}_N\_\text{MIN}\_\text{EXP} \\
\text{DEC}_N\_\text{MIN}\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MIN}\_10\_\text{EXP} \\
\text{FLT}_N\_\text{MIN}\_10\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MAX}\_\text{EXP} \\
\text{DEC}_N\_\text{MAX}\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MAX}\_10\_\text{EXP} \\
\text{FLT}_N\_\text{MAX}\_10\_\text{EXP}
\]

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

\[
\text{FLT}_N\_\text{MAX} \\
\text{DEC}_N\_\text{MAX}
\]

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

\[
\text{FLT}_N\_\text{EPSILON} \\
\text{DEC}_N\_\text{EPSILON}
\]

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

\[
\text{FLT}_N\_\text{MIN} \\
\text{DEC}_N\_\text{MIN}
\]

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

\[
\text{FLT}_N\_\text{TRUE}\_\text{MIN} \\
\text{DEC}_N\_\text{TRUE}\_\text{MIN}
\]
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 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, 7.12.11b.1, X.12.11b.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, 7.12.11b.2, 7.12.11b.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. (7.22.1.3, 7.22.1.3).

— 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. (7.22.1.2a) (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 ≥ 32 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:

\[ fN \ F N \ fNx \ FNx \ dN \ DN \ dNx \ DNx \]

[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 constant of decimal floating type is the same as for the result value of the corresponding \texttt{strtodN} or \texttt{strtodNx} function for the same numeric string.

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 5.2.4.2.2a).

[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:

\begin{align*}
&_\text{Float}N, \text{where } N \text{ is } 16, 32, 64, \text{ or } \geq 128 \text{ and a multiple of } 32 \\
&_\text{Float32x} \\
&_\text{Float64x} \\
&_\text{Float128x} \\
&_\text{Decimal}N, \text{where } N \geq 32 \text{ and a multiple of } 32 \\
&_\text{Decimal64x} \\
&_\text{Decimal128x}
\end{align*}

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

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

\begin{itemize}
  \item \_FloatN, \text{where } N \text{ is } 16, 32, 64, \text{ or } \geq 128 \text{ and a multiple of } 32
  \item \_Float32x
  \item \_Float64x
  \item \_Float128x
  \item \_DecimalN, \text{where } N \geq 32 \text{ and a multiple of } 32
  \item \_Decimal64x
  \item \_Decimal128x
  \item \_FloatN \_Complex, \text{where } N \text{ is } 16, 32, 64, \text{ or } \geq 128 \text{ and a multiple of } 32
  \item \_Float32x \_Complex
  \item \_Float64x \_Complex
  \item \_Float128x \_Complex
\end{itemize}

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 \texttt{<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 \texttt{<float.h>} is first included:
for supported types _FloatN:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT/N_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/N_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/N_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/N_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/N_MIN</td>
<td></td>
</tr>
<tr>
<td>FLT/N_MAX</td>
<td></td>
</tr>
<tr>
<td>FLT/N_MIN_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/N_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/N_EPSILON</td>
<td></td>
</tr>
<tr>
<td>FLT/N_MIN_EXP_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/N_TRUE_MIN</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT/N_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/N_DIG</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT/NX_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_DECIMAL_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_DIG</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_MAX</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_MIN_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_MAX_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_EPSILON</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_MIN_EXP_10_EXP</td>
<td></td>
</tr>
<tr>
<td>FLT/NX_TRUE_MIN</td>
<td></td>
</tr>
<tr>
<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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC/N_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>DEC/N_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>DEC/N_MAX_EXP</td>
<td></td>
</tr>
<tr>
<td>DEC/N_MAX</td>
<td></td>
</tr>
<tr>
<td>DEC/N_MIN</td>
<td></td>
</tr>
<tr>
<td>DEC/N_MAX</td>
<td></td>
</tr>
<tr>
<td>DEC/N_EPSILON</td>
<td></td>
</tr>
<tr>
<td>DEC/N_TRUE_MIN</td>
<td></td>
</tr>
</tbody>
</table>

for supported types _DecimalNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC/NX_MANT_DIG</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_MIN_EXP</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_MAX_EXP</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_MAX</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_MIN</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_MAX</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_EPSILON</td>
<td></td>
</tr>
<tr>
<td>DEC/NX_TRUE_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:

25 for supported types _FloatN:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfN</td>
<td></td>
</tr>
<tr>
<td>casinfN</td>
<td></td>
</tr>
<tr>
<td>csinfN</td>
<td></td>
</tr>
<tr>
<td>ccosfN</td>
<td></td>
</tr>
<tr>
<td>cctanfN</td>
<td></td>
</tr>
<tr>
<td>ccsinfN</td>
<td></td>
</tr>
<tr>
<td>cctanfN</td>
<td></td>
</tr>
<tr>
<td>cacoshfN</td>
<td></td>
</tr>
<tr>
<td>casinhfN</td>
<td></td>
</tr>
<tr>
<td>csqrtfN</td>
<td></td>
</tr>
<tr>
<td>cargfN</td>
<td></td>
</tr>
<tr>
<td>cimagfN</td>
<td></td>
</tr>
<tr>
<td>CMPLXFN</td>
<td></td>
</tr>
<tr>
<td>conjfN</td>
<td></td>
</tr>
<tr>
<td>crealfN</td>
<td></td>
</tr>
<tr>
<td>cprojfN</td>
<td></td>
</tr>
</tbody>
</table>

30 for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfNx</td>
<td></td>
</tr>
<tr>
<td>casinfNx</td>
<td></td>
</tr>
<tr>
<td>ccosfNx</td>
<td></td>
</tr>
<tr>
<td>ccsinfNx</td>
<td></td>
</tr>
<tr>
<td>cctanfNx</td>
<td></td>
</tr>
<tr>
<td>cctanfNx</td>
<td></td>
</tr>
<tr>
<td>cacoshfNx</td>
<td></td>
</tr>
<tr>
<td>casinhfNx</td>
<td></td>
</tr>
<tr>
<td>csqrtfNx</td>
<td></td>
</tr>
<tr>
<td>cargfNx</td>
<td></td>
</tr>
<tr>
<td>cimagfNx</td>
<td></td>
</tr>
</tbody>
</table>

40 for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfNx</td>
<td></td>
</tr>
<tr>
<td>casinfNx</td>
<td></td>
</tr>
<tr>
<td>ccosfNx</td>
<td></td>
</tr>
<tr>
<td>ccsinfNx</td>
<td></td>
</tr>
<tr>
<td>cctanfNx</td>
<td></td>
</tr>
<tr>
<td>cctanfNx</td>
<td></td>
</tr>
<tr>
<td>cacoshfNx</td>
<td></td>
</tr>
<tr>
<td>casinhfNx</td>
<td></td>
</tr>
<tr>
<td>csqrtfNx</td>
<td></td>
</tr>
<tr>
<td>cargfNx</td>
<td></td>
</tr>
<tr>
<td>cimagfNx</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfN</td>
<td></td>
</tr>
<tr>
<td>casinfN</td>
<td></td>
</tr>
<tr>
<td>ccosfN</td>
<td></td>
</tr>
<tr>
<td>ccsinfN</td>
<td></td>
</tr>
<tr>
<td>cctanfN</td>
<td></td>
</tr>
<tr>
<td>cctanfN</td>
<td></td>
</tr>
<tr>
<td>cacoshfN</td>
<td></td>
</tr>
<tr>
<td>casinhfN</td>
<td></td>
</tr>
</tbody>
</table>

50 for supported types _DecimalNx:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cacosfN</td>
<td></td>
</tr>
<tr>
<td>casinfN</td>
<td></td>
</tr>
<tr>
<td>ccosfN</td>
<td></td>
</tr>
<tr>
<td>ccsinfN</td>
<td></td>
</tr>
<tr>
<td>cctanfN</td>
<td></td>
</tr>
<tr>
<td>cctanfN</td>
<td></td>
</tr>
<tr>
<td>cacoshfN</td>
<td></td>
</tr>
<tr>
<td>casinhfN</td>
<td></td>
</tr>
</tbody>
</table>
X.8.3 `<math.h>`

[1] The following identifiers are defined or declared only if

```c
__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`:

```c
log1pfN fromfpfN
log2fN ufromfpfN
logbfN fromfpfxN
modfN ufromfpfxN
acoslN scalblnfN remainderfN
asinfN scalblnfN remainderfN
atanfN cbrtfN remquofN
atan2fN fabsfN copysignfN
coslN hypotfN nanfN
sinfN powfN nextafterfN
tanfN sqrtfN nextupfN
acoshfN erfN nextdownfN
asinhfN erfcN canonicalizefN
atanhfN lgammafN encodefN
coshfN tgammafN decodefN
sinhfN ceilfN fdimfN
tanhfN floorfN fmaxfN
expfN nearbyintfN fminfN
exp2fN rintfN fmaxmagfN
expmfN lrintfN fminmagfN
frexpN llrintfN fmafN
ilogbfN roundfN totalorderfN
ldexpfN lroundfN totalordermagfN
1logbfN lroundfN getpayloadfN
logfN truncfN setpayloadfN
log10fN roundevenfN setpayloads sigfN
```

for supported types `_FloatNx`:

```c
log1pfNx fromfpfNx
log2fNx ufromfpfNx
logbfNx fromfpfxN
modfNx ufromfpfxN
acoslfNx scalblnfN remainderfNx
asinfNx scalblnfN remainderfNx
atanfNx cbrtfN remquofNx
cosfNx fabsfNx remquofNx
atan2fNx scalblnfN remainderfNx
cosfNx scalblnfN remainderfNx
sinfNx hypotfN copysignfN
tanfNx powfN nanfN
cosfNx sqrtfN nextafterfNx
acoshfNx erffN nextupfN
asinhfNx erffN nextupfN
```
atanhfNx erfclfNx nextdownfNx
expfNx lgammafNx canonicalizefNx
coshfNx tgammafNx fdimfNx
sinhfNx ceilfNx fmaxfNx
tanhfNx floorfNx fminfNx
exp2fNx nearbyintfNx fmaxmagfNx
expmlfNx rintfNx fminmagfNx
frexpfnX lrintfNx fmafNx
ilogbfNx llrintfNx totalorderfNx
llogbfNx roundfNx totalordermagfNx
ldexpfNx lroundfNx getpayloadfNx
logfNx llroundfNx setpayloadf Nx
log10fNx truncfNx setpayloadsigfNx

for supported types _FloatM and _FloatN where \( M < N \):

15
FP_FAST_FMADDFN FP_FAST_FMFMADFN fMmulfN
FP_FAST_FMSUBFNN FP_FAST_FMSQRTFNN fMdivfN
FP_FAST_FMMULFN fMaddfN fMfmafN
FP_FAST_FMDIVFNN fMsubfN fMsqrtfN

for supported types _FloatM and _FloatN where \( M \leq N \):

20
FP_FAST_FMADDFNN FP_FAST_FMFMAFNX fMmulfN
FP_FAST_FMSUBFNN FP_FAST_FMSQRTFNNX fMdivfN
FP_FAST_FMMULFNX fMaddfN fMfmafN
FP_FAST_FMDIVFNNX fMsubfN fMsqrtfN

for supported types _FloatMx and _FloatN where \( M < N \):

25
FP_FAST_FMXADDFN FP_FAST_FMXMFMAFNX fMxmulfN
FP_FAST_FMXSUBFNN FP_FAST_FMXSQRTFNNX fMxdivfN
FP_FAST_FMXMULFNX fMxaddfN fMxfmafN
FP_FAST_FMXDIVFNNX fMxsubfN fMxsqrtfN

for supported types _FloatMx and _FloatN where \( M < N \):

30
FP_FAST_FMXADDFNX FP_FAST_FMXMFMAFX fMxmulfN
FP_FAST_FMXSUBFX FP_FAST_FMXSQRTFX fMxdivfN
FP_FAST_FMXMULFNX fMxaddfN fMxfmafN
FP_FAST_FMXDIVFX fMxsubfN fMxsqrtfN

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

fMenCFN

for supported types _DecimalN, where \( N \neq 32, 64, \) and 128:

_DecimalN_t logbdN fmoddN
HUGE_VAL_DN modfD N remainderDN
SNANDN scalbnDN copysignD N
FP_FAST_FMU DN scalblndD N nandN
acosDN cbrtdN nextafterDN
asindN fabsdN nextupdN
<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>atan2dN</td>
<td>hypotdN</td>
<td>nextdowndN</td>
</tr>
<tr>
<td>cosdN</td>
<td>powdN</td>
<td>canonicalizedN</td>
</tr>
<tr>
<td>sindN</td>
<td>sqrtdN</td>
<td>quantizedN</td>
</tr>
<tr>
<td>tandN</td>
<td>erfN</td>
<td>samequantumdN</td>
</tr>
<tr>
<td>acoshdN</td>
<td>lgammadN</td>
<td>llquantexpdN</td>
</tr>
<tr>
<td>asinhN</td>
<td>tgammaN</td>
<td>encoinedcdN</td>
</tr>
<tr>
<td>atanhdN</td>
<td>ceildN</td>
<td>decodedcdN</td>
</tr>
<tr>
<td>coshdN</td>
<td>floordN</td>
<td>encodbindN</td>
</tr>
<tr>
<td>5</td>
<td>erfcdN</td>
<td>quantudN</td>
</tr>
<tr>
<td>sinhN</td>
<td>nearbyintdN</td>
<td>decodebindN</td>
</tr>
<tr>
<td>tanhdN</td>
<td>rintdn</td>
<td>fdimdN</td>
</tr>
<tr>
<td>expN</td>
<td>lrintdN</td>
<td>fmadN</td>
</tr>
<tr>
<td>exp2dN</td>
<td>llrintdN</td>
<td>llquantexpdN</td>
</tr>
<tr>
<td>expm1dN</td>
<td>rounddN</td>
<td>fmaxmagnD</td>
</tr>
<tr>
<td>10</td>
<td>lrounddN</td>
<td>fminmagnD</td>
</tr>
<tr>
<td>frexpN</td>
<td>lrounddN</td>
<td>fmadN</td>
</tr>
<tr>
<td>15</td>
<td>lrounddN</td>
<td>totalorderdN</td>
</tr>
<tr>
<td>logbdN</td>
<td>truncdN</td>
<td>totalordermagnD</td>
</tr>
<tr>
<td>ldexpdN</td>
<td>roundvdN</td>
<td>getpayloaddN</td>
</tr>
<tr>
<td>20</td>
<td>fromfpdN</td>
<td>setpayloaddN</td>
</tr>
<tr>
<td>log10dN</td>
<td>ufromfpdN</td>
<td>setpayloaddN</td>
</tr>
<tr>
<td>log1pdN</td>
<td>fromfpxedN</td>
<td>setpayloaddN</td>
</tr>
<tr>
<td>log2dN</td>
<td>ufromfpdN</td>
<td>setpayloaddN</td>
</tr>
</tbody>
</table>

for supported types _DecimalN_x:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_D_Nx</td>
<td>log2d_Nx</td>
<td>ufromfpd_Nx</td>
</tr>
<tr>
<td>25</td>
<td>logbd_Nx</td>
<td>fromfpd_Nx</td>
</tr>
<tr>
<td>SNAND_Nx</td>
<td>modfd_Nx</td>
<td>ufromfpxd_Nx</td>
</tr>
<tr>
<td>FP_FAST_FMAD_Nx</td>
<td>scalbnd_Nx</td>
<td>fmodd_Nx</td>
</tr>
<tr>
<td>acosd_Nx</td>
<td>scalbld_Nx</td>
<td>remainderd_Nx</td>
</tr>
<tr>
<td>asind_Nx</td>
<td>cbrtd_Nx</td>
<td>copyssignd_Nx</td>
</tr>
<tr>
<td>30</td>
<td>fabsd_Nx</td>
<td>nand_Nx</td>
</tr>
<tr>
<td>cosd_Nx</td>
<td>hypotd_Nx</td>
<td>nextafterd_Nx</td>
</tr>
<tr>
<td>sind_Nx</td>
<td>powd_Nx</td>
<td>nextupd_Nx</td>
</tr>
<tr>
<td>tand_Nx</td>
<td>sqrttd_Nx</td>
<td>nextdownd_Nx</td>
</tr>
<tr>
<td>acoshd_Nx</td>
<td>erfN</td>
<td>canonicalized_Nx</td>
</tr>
<tr>
<td>35</td>
<td>erfcdN</td>
<td>quantized_Nx</td>
</tr>
<tr>
<td>asinhd_Nx</td>
<td>lgammaN</td>
<td>samequantumd_Nx</td>
</tr>
<tr>
<td>atanhd_Nx</td>
<td>tgammaN</td>
<td>quantumd_Nx</td>
</tr>
<tr>
<td>coshd_Nx</td>
<td>ceildN</td>
<td>llquantexpd_Nx</td>
</tr>
<tr>
<td>sinhd_Nx</td>
<td>floordN</td>
<td>fdimd_Nx</td>
</tr>
<tr>
<td>tanhd_Nx</td>
<td>nearbyintd_Nx</td>
<td>fmaxd_Nx</td>
</tr>
<tr>
<td>40</td>
<td>rintd_Nx</td>
<td>fminmagn_Nx</td>
</tr>
<tr>
<td>expN</td>
<td>lrintd_Nx</td>
<td>fmad_Nx</td>
</tr>
<tr>
<td>expm1dN</td>
<td>roundd_Nx</td>
<td>fmaxmagn_Nx</td>
</tr>
<tr>
<td>frexpN</td>
<td>llrintd_Nx</td>
<td>fminmagn_Nx</td>
</tr>
<tr>
<td>45</td>
<td>roundvd_Nx</td>
<td>getpayloadd_Nx</td>
</tr>
<tr>
<td>logbdN</td>
<td>truncd_Nx</td>
<td>totalorderd_Nx</td>
</tr>
<tr>
<td>ldexpdN</td>
<td>llroundd_Nx</td>
<td>totalordermagnd_Nx</td>
</tr>
<tr>
<td>logd_Nx</td>
<td>truncated_Nx</td>
<td>setpayloadd_Nx</td>
</tr>
<tr>
<td>log10d_Nx</td>
<td>roundvd_Nx</td>
<td>setpayloadd_Nx</td>
</tr>
<tr>
<td>log1pdN</td>
<td>fromfpd_Nx</td>
<td>setpayloadd_Nx</td>
</tr>
</tbody>
</table>

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

\begin{verbatim}
FP_FAST_DMAADDN \quad \text{FP\_FAST\_D\_FMAD}\ N \quad \text{dMmuld}\ N
FP_FAST_DMSUBD\ N \quad \text{FP\_FAST\_D\_MSQRTD}\ N \quad \text{dMdivd}\ N
FP_FAST_DMULDN \quad \text{dMaddd}\ N \quad \text{dMfmad}\ N
FP_FAST_DMDIVD\ N \quad \text{dMsubd}\ N \quad \text{dMsqrtd}\ N
\end{verbatim}

for supported types \texttt{DecimalM} and \texttt{DecimalNx} where \( M \leq N \):

\begin{verbatim}
FP_FAST_DMAADDNX \quad \text{FP\_FAST\_D\_FMAD}\NX \quad \text{dMmuld}\Nx \quad \text{dMdivd}\Nx
FP_FAST_DMSUBDX \quad \text{FP\_FAST\_D\_MSQRTDX}\Nx \quad \text{dMdivd}\Nx
\end{verbatim}

for supported types \texttt{DecimalMx} and \texttt{DecimalN} where \( M < N \):

\begin{verbatim}
FP_FAST_DMXADDNX \quad \text{FP\_FAST\_D\_FXMAD}\NX \quad \text{dMmuld}\Nx \quad \text{dMdivd}\Nx
FP_FAST_DMSUBDX \quad \text{FP\_FAST\_D\_MSQRTDX}\Nx \quad \text{dMdivd}\Nx
\end{verbatim}

for supported types \texttt{DecimalMx} and \texttt{DecimalNx} where \( M < N \):

\begin{verbatim}
FP_FAST_DMXADDNX \quad \text{FP\_FAST\_D\_FXMAD}\NX \quad \text{dMmuld}\Nx \quad \text{dMdivd}\Nx
FP_FAST_DMSUBDX \quad \text{FP\_FAST\_D\_MSQRTDX}\Nx \quad \text{dMdivd}\Nx
\end{verbatim}

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

\begin{verbatim}
dMex\ N \quad \text{dMencdecd}\ N \quad \text{dMencb}\ N
\end{verbatim}

25 X.8.4 \texttt{<stdlib.h>}

[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{<stdlib.h>} is first included:

for supported types \texttt{FloatN}:

\begin{verbatim}
strfromf\ N \quad \text{strtof}\ N
\end{verbatim}

30 for supported types \texttt{FloatNx}:

\begin{verbatim}
strfromf\Nx \quad \text{strtof}\Nx
\end{verbatim}

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

\begin{verbatim}
strfromd\ N \quad \text{strtod}\ N
\end{verbatim}

for supported types \texttt{DecimalNx}:

\begin{verbatim}
strfromd\Nx \quad \text{strtod}\Nx
\end{verbatim}
for supported IEC 60559 arithmetic and non-arithmetic binary interchange formats of width $N$:

$$\text{strfromencf}_N \quad \text{strtoencf}_N$$

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

$$\text{strfromencdec}_N \quad \text{strtoencdec}_N$$
$$\text{strfromencbind}_N \quad \text{strtoencbind}_N$$

X.9 Complex arithmetic <complex.h>

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

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

[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

$$\text{_Float}_N \text{ complex cacosf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex cacosf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex casinf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex casinf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex catanf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex catanf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex ccosf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex ccosf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex csinf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex csinf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex ctanf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex ctanf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

7.3.6 Hyperbolic functions

$$\text{_Float}_N \text{ complex cacoshf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex cacoshf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex casinhf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex casinhf}_{Nx}(_\text{Float}_N \text{ complex } z);$$

$$\text{_Float}_N \text{ complex catanhf}_N(_\text{Float}_N \text{ complex } z);$$
$$\text{_Float}_N \text{ complex catanhf}_{Nx}(_\text{Float}_N \text{ complex } z);$$
_FloatN complex ccoshf(_FloatN complex z);
_FloatNx complex ccoshfNx(_FloatNx complex z);

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

_FloatN complex ctanhf(_FloatN complex z);
_FloatNx complex ctanhfNx(_FloatNx complex z);

7.3.7 Exponential and logarithmic functions

_FloatN complex cexpf(_FloatN complex z);
_FloatNx complex cexpfNx(_FloatNx complex z);

_FloatN complex clogf(_FloatN complex z);
_FloatNx complex clogfNx(_FloatNx complex z);

7.3.8 Power and absolute value functions

_FloatN cabsf(_FloatN complex z);
_FloatNx cabsfNx(_FloatNx complex z);

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

_FloatN complex csqrfN(_FloatN complex z);
_FloatNx complex csqrfNx(_FloatNx complex z);

7.3.9 Manipulation functions

_FloatN cargf(_FloatN complex z);
_FloatNx cargfNx(_FloatNx complex z);

_FloatN cimagf(_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 (7.6.1a)

```c
#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 7.6.1a, 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 listed in the table indicates the family of functions of all standard and binary floating types (for example, `acosf, acosl, acosfN`, and `acosfNx` as well as `acos`). The `fMencfN, strfromencfN`, and `strtoencfN` functions for binary interchange types are also affected by constant rounding modes.

[4] In 7.6.1b, in the table of functions affected by constant rounding modes for decimal floating types, each `<math.h>` function indicates the family of functions of all decimal floating types (for example, `acosdNx`, as well as `acosdN`). The `dMencbindN, dMentdecN, strfromencbindN, strfromenctdecN, strtoencbindN, and strtoenctdecN` functions for decimal interchange types are also affected by constant rounding modes.

[5] The `fegetround` function (7.6.3.1) gets the current value of the dynamic rounding direction mode for operations for standard and binary floating types. The `fesetround` function (7.6.3.2) 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 (7.12.14) 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

```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_ (X.3) 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_ (X.3) to be used for evaluating operations and constants of that decimal floating type.

### X.11.1 Macros

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

[2] The macros

```c
HUGE_VAL_F
HUGE_VAL_D
HUGE_VAL_FN
HUGE_VAL_DN
```

expand to constant expressions of types _FloatN, _DecimalN, _FloatNx, and _DecimalNx, respectively, representing positive infinity.

[3] The signaling NaN macros

```c
SNANFN
SNANDN
SNANFNX
SNANDNX
```

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

\[
\begin{align*}
\text{FP_FAST_FMAF}N \\
\text{FP_FAST_FMAD}N \\
\text{FP_FAST_FMAF}N X \\
\text{FP_FAST_FMAD}N X
\end{align*}
\]

are, respectively, _FloatN, _DecimalN, _FloatNx, and _DecimalNx analogues of \text{FP_FAST_FMA}.

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

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

\[
\begin{align*}
\text{FP_FAST_FMADDF}N \\
\text{FP_FAST_FMSUBF}N \\
\text{FP_FAST_FMULFN} \\
\text{FP_FAST_FMIVFN} \\
\text{FP_FAST_FMMAFN} \\
\text{FP_FAST_FMSQRTF}N \\
\text{FP_FAST_DMMADD}N \\
\text{FP_FAST_DMSUBD}N \\
\text{FP_FAST_DMMULD}N \\
\text{FP_FAST_DMIDVD}N \\
\text{FP_FAST_DMFAD}N \\
\text{FP_FAST_DMSQRTD}N
\end{align*}
\]

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

\[
\begin{align*}
\text{FP_FAST_FMADD}N X \\
\text{FP_FAST_FMSUB}N X \\
\text{FP_FAST_FMUL}N X \\
\text{FP_FAST_FDIV}N X \\
\text{FP_FAST_FMAD}N X \\
\text{FP_FAST_FMSQRT}N X \\
\text{FP_FAST_DMADD}N X \\
\text{FP_FAST_DMSUB}N X \\
\text{FP_FAST_DMUL}N X \\
\text{FP_FAST_DIV}N X \\
\text{FP_FAST_DMFAD}N X \\
\text{FP_FAST_DMSQRT}N X
\end{align*}
\]

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

```
FP_FAST_FMXADDFN
FP_FAST_FMXSUBFN
FP_FAST_FMXMULFN
FP_FAST_FMXDIVFN
FP_FAST_FMXFMAFN
FP_FAST_FMXSQRTPN
FP_FAST_DMXADDDN
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_FMXDIVFXN
FP_FAST_FMXFMAFXN
FP_FAST_FMXSQRTPFXN
FP_FAST_DMXADDDNX
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);
```
7.12.5 Hyperbolic functions

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

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

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

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

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

_FloatN 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 atanhN(_DecimalN x);
_DecimalNx atanhN(_DecimalNx x);

_FloatN coshN(_FloatN x);
_FloatNx coshN(_FloatNx x);
_DecimalN coshdN(_DecimalN x);
_DecimalNx coshdNx(_DecimalNx x);

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

7.12.6 Exponential and logarithmic functions

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

_FloatN 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 frexpdnx(_DecimalNx value, int *exp);

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

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

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

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

_FloatN log10fN(_FloatN x);
_FloatNx log10fNx(_FloatNx x);
DecimalN log10dN(_DecimalN x);
DecimalNx log10dNx(_DecimalNx x);
_FloatN log1pf(_FloatN x);
_FlottN log1pfN(_FloatN x);
DecimalN log1pdN(_DecimalN x);
_FlottN log1pdN(_DecimalN x);

_FloatN log2f(_FloatN x);
_FloatN log2fN(_FloatN x);
DecimalN log2dN(_DecimalN x);
_FloatN log2dN(_DecimalN x);

_FloatN logbf(_FloatN x);
_FloatN logbfN(_FloatN x);
DecimalN logbdN(_DecimalN x);
_FloatN logbdN(_DecimalN x);

_FloatN modff(_FloatN x, _FloatN *iptr);
_FloatN modffN(_FloatN x, _FloatN *iptr);
DecimalN modfdN(_DecimalN x, _DecimalN *iptr);
_FloatN modfdN(_DecimalN x, _DecimalN *iptr);

_FloatN scalbnf(_FloatN value, int exp);
_FloatN scalbnfN(_FloatN value, int exp);
DecimalN scalbndN(_DecimalN value, int exp);
_FloatN scalbndN(_DecimalN value, int exp);

_FloatN scalblnf(_FloatN value, long int exp);
_FloatN scalblnfN(_FloatN value, long int exp);
DecimalN scalblndN(_DecimalN value, long int exp);
_FloatN scalblndN(_DecimalN value, long int exp);

_FloatN cbrtf(_FloatN x);
_FloatN cbrtfN(_FloatN x);
DecimalN cbrtfN(_DecimalN x);
_FloatN cbrtfN(_DecimalN x);

_FloatN fabsf(_FloatN x);
_FloatN fabsfN(_FloatN x);
DecimalN fabsdN(_DecimalN x);
_FloatN fabsdN(_DecimalN x);

_FloatN hypotf(_FloatN x, _FloatN y);
_FloatN hypotfN(_FloatN x, _FloatN y);
DecimalN hypotdN(_DecimalN x, _DecimalN y);
_FloatN hypotdN(_DecimalN x, _DecimalN y);

_FloatN powf(_FloatN x, _FloatN y);
_FloatN powfN(_FloatN x, _FloatN y);
DecimalN powdN(_DecimalN x, _DecimalN y);
_FloatN powdN(_DecimalN x, _DecimalN y);

_FloatN sqrtf(_FloatN x);
_FloatN sqrtfN(_FloatN x);

7.12.7 Power and absolute-value functions
7.12.8 Error and gamma functions

_FloatN erffN(_FloatN x);
_FloatNx erffNx(_FloatNx x);
.DecimalN erfN(_DecimalN x);
.DecimalNx erfNx(_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 rintlN(_DecimalN x);
.DecimalNx rintlNx(_DecimalNx x);

long int lrintfN(_FloatN x);
long int lrintfNx(_FloatNx x);
long int lrintlN(_DecimalN x);
long int lrintlNx(_DecimalNx x);
long long int llrintfN(_FloatN x);
long long int llrintfNx(_FloatNx x);
long long int llrintdN(_DecimalN x);
long long int llrintdNx(_DecimalNx x);

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

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

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

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

intmax_t fromfpfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpfNx(_FloatNx x, int round, unsigned int width);
intmax_t fromfpdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpdNx(_DecimalNx x, int round, unsigned int width);

uintmax_t ufromfpfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpfNx(_FloatNx x, int round, unsigned int width);
uintmax_t ufromfpdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpdNx(_DecimalNx x, int round, unsigned int width);

intmax_t fromfpxfN(_FloatN x, int round, unsigned int width);
intmax_t fromfpxfNx(_FloatNx x, int round, unsigned int width);
intmax_t fromfpxdN(_DecimalN x, int round, unsigned int width);
intmax_t fromfpxdNx(_DecimalNx x, int round, unsigned int width);

uintmax_t ufromfpxfN(_FloatN x, int round, unsigned int width);
uintmax_t ufromfpxfNx(_FloatNx x, int round, unsigned int width);
uintmax_t ufromfpxdN(_DecimalN x, int round, unsigned int width);
uintmax_t ufromfpxdNx(_DecimalNx x, int round, unsigned int width);
7.12.10 Remainder functions

```c
_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

```c
_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 quantumdN(_DecimalN x);
.DecimalNx quantumdNx(_DecimalNx x);
```
7.12.12 Maximum, minimum, and positive difference functions

```c
_FloatN fdimfN(_FloatN x, _FloatN y);
_FloatN fdimfNx(_FloatNx x, _FloatNx y);
.DecimalN fdimdN(_DecimalN x, _DecimalN y);
.DecimalNx fdimdNx(_DecimalNx x, _DecimalNx y);

_FloatN fmaxfN(_FloatN x, _FloatN y);
_FloatN fmaxfNx(_FloatNx x, _FloatNx y);
.DecimalN fmaxdN(_DecimalN x, _DecimalN y);
.DecimalNx fmaxdNx(_DecimalNx x, _DecimalNx y);

_FloatN fminfN(_FloatN x, _FloatN y);
_FloatN fminfNx(_FloatNx x, _FloatNx y);
.DecimalN fminNdN(_DecimalN x, _DecimalN y);
.DecimalNx fminNdNx(_DecimalNx x, _DecimalNx y);

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

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

7.12.13 Floating multiply-add

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

7.12.13a Functions that round result to narrower type

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

_FloatM fMsubfN(_FloatN x, _FloatN y); // M < N
_FloatM fMsubfNx(_FloatNx x, _FloatNx y); // M <= N
_FloatM fMxfsubfN(_FloatN x, _FloatN y); // M < N
_FloatM fMxfsubfNx(_FloatNx x, _FloatNx y); // M < N

_FloatM fMmulfN(_FloatN x, _FloatN y); // M < N
_FloatM fMmulfNx(_FloatNx x, _FloatNx y); // M <= N
_FloatM fMxmulfN(_FloatN x, _FloatN y); // M < N
_FloatM fMxmulfNx(_FloatNx x, _FloatNx y); // M < N

_FloatM fMdivfN(_FloatN x, _FloatN y); // M < N
_FloatM fMdivfNx(_FloatNx x, _FloatNx y); // M <= N
_FloatM fMxdivfN(_FloatN x, _FloatN y); // M < N
_FloatM fMxdivfNx(_FloatNx x, _FloatNx y); // M < N

_FloatM fMfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMfmafNx(_FloatNx x, _FloatNx y, _FloatNx z); // M <= N
_FloatM fMxfmafN(_FloatN x, _FloatN y, _FloatN z); // M < N
_FloatM fMxfmafNx(_FloatNx x, _FloatNx y, _FloatNx z); // M < N

_FloatM fMsqrtfN(_FloatN x); // M < N
_FloatM fMsqrtfNx(_FloatNx x); // M <= N
_FloatM fMxsqrtfN(_FloatN x); // M < N
_FloatM fMxsqrtfNx(_FloatNx x); // M < N
\_DecimalM dMsqrt\_dN(_\_DecimalN x);  // M < N
\_DecimalM dMsqrt\_dNx(_\_DecimalNx x);  // M <= N
\_DecimalMx dMxsqrt\_dN(_\_DecimalN x);  // M < N
\_DecimalMx dMxsqrt\_dNx(_\_DecimalNx x);  // M < N

F.10.12 Total order functions

int totalorder\_fN(_\_FloatN x, _\_FloatN y);
int totalorder\_fNx(_\_FloatNx x, _\_FloatN y);
int totalorder\_dN(_\_DecimalN x, _\_DecimalN y);
int totalorder\_dNx(_\_DecimalNx x, _\_DecimalNx y);

int totalorder\_mag\_fN(_\_FloatN x, _\_FloatN y);
int totalorder\_mag\_fNx(_\_FloatNx x, _\_FloatN y);
int totalorder\_mag\_dN(_\_DecimalN x, _\_DecimalN y);
int totalorder\_mag\_dNx(_\_DecimalNx x, _\_DecimalNx y);

F.10.13 Payload functions

_\_FloatN getpayload\_fN(const _\_FloatN *x);
_\_FloatNx getpayload\_fNx(const _\_FloatNx *x);
_\_DecimalN getpayload\_dN(const _\_DecimalN *x);
_\_DecimalNx getpayload\_dNx(const _\_DecimalNx *x);

int setpayload\_fN(_\_FloatN *res, _\_FloatN pl);
int setpayload\_fNx(_\_FloatNx *res, _\_FloatN pl);
int setpayload\_dN(_\_DecimalN *res, _\_DecimalN pl);
int setpayload\_dNx(_\_DecimalNx *res, _\_DecimalNx pl);

int setpayload\_sig\_fN(_\_FloatN *res, _\_FloatN pl);
int setpayload\_sig\_fNx(_\_FloatNx *res, _\_FloatN pl);
int setpayload\_sig\_dN(_\_DecimalN *res, _\_DecimalN pl);
int setpayload\_sig\_dNx(_\_DecimalNx *res, _\_DecimalNx pl);

[2] The specification of the frexp functions (7.12.6.4) 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^{\exp}.

[3] The specification of the ldexp functions (7.12.6.6) applies to the functions for binary floating types like those for standard floating types: they return x \times 2^{\exp}.


[5] The specification of the scalbn and scalbln functions (7.12.6.13) 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

[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

[1] #include <math.h>
   void encodefN(unsigned char * restrict encptr,
                  const _FloatN * restrict xptr);

Description

[2] 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


X.11.3.1.2 The decodefN functions

Synopsis

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

Description

[2] 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


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. An implementation shall provide both dMencodedN and dMencbindN 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 $d_{MencdecdN}$ and $d_{MencbindN}$ functions

Synopsis

[1] #include <math.h>
    void $d_{MencdecdN}$ (unsigned char * restrict enc$M$ptr,
    const unsigned char * restrict enc$N$ptr);
    void $d_{MencbindN}$ (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 $d_{MencdecdN}$ functions convert between formats using the encoding scheme
based on decimal encoding of the significand. The $d_{MencbindN}$ 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.
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.

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**

This subclause expands 7.22.1.2a and 7.22.1.2b to also include functions for the interchange and extended floating types. It adds to the synopsis in 7.22.1.2a the prototypes

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

It encompasses the prototypes in 7.22.1.2b 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);
```

The descriptions and returns for the added functions are analogous to the ones in 7.22.1.2a and 7.22.1.2b.

**X.12.2 String to floating**

This subclause expands 7.22.1.3 and 7.22.1.3a to also include functions for the interchange and extended floating types. It adds to the synopsis in 7.22.1.3 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 7.22.1.3a by replacing them with

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

The descriptions and returns for the added functions are analogous to the ones in 7.22.1.3 and 7.22.1.3a.

**X.12.3 String from encoding**

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 `strfromencecdN` and `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 strfromencf functions

Synopsis

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

Description

[2] The strfromencf functions are similar to the strfromf 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.

Returns

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

X.12.3.2 The strfromencdecd and strfromencbind functions

Synopsis

[1] #include <stdlib.h>
   int strfromencdecdN(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);

Description

[2] The strfromencdecd functions are similar to the strfromd 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 strfromencbind functions are similar to the strfromd 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.

Returns

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

X.12.4 String to encoding

X.12.4.1 The strtoencf functions

Synopsis

[1] #include <stdlib.h>
   void strtoencfN(unsigned char * restrict encptr,
                   const char * restrict nptr, char ** restrict endptr);
Description

[2] The \texttt{strtoencfN} functions are similar to the \texttt{strtofN} functions, except they store an IEC 60559 encoding of the result as an \(N/8\) element array in the object pointed to by \texttt{encptr}. The order of bytes in the arrays is implementation-defined.

5

Returns

[3] These functions return no value.

X.12.4.2 The \texttt{strtoencdecdN} and \texttt{strtoencbindN} functions

Synopsis

[1] \#include <stdlib.h>

\begin{verbatim}
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);
\end{verbatim}

Description

[2] The \texttt{strtoencdecdN} and \texttt{strtoencbindN} functions are similar to the \texttt{strtodN} functions, except they store an IEC 60559 encoding of the result as an \(N/8\) element array in the object pointed to by \texttt{encptr}. The \texttt{strtoencdecdN} functions produce an encoding in the encoding scheme based on decimal encoding of the significand. The \texttt{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 \texttt{<tgmath.h>}

[1] This clause enhances the specification of type-generic macros in \texttt{<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 \texttt{carg, cimag, conj, cproj}, or \texttt{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 \texttt{f} or \texttt{d} prefix are (as in 7.25):

\begin{verbatim}
fadd fmul ffma
dadd dmul dfma
fsub fdiv fsqrt
dsub ddiv dsqrt
\end{verbatim}

and the macros with \texttt{fM}, \texttt{fMx}, \texttt{dM}, or \texttt{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>Prefix</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{f}</td>
<td>\texttt{double}</td>
</tr>
</tbody>
</table>
| \texttt{d} | \texttt{long
double} |
| \texttt{fM} | \texttt{_FloatMx} if supported, else \texttt{_FloatN} for minimum \texttt{N > M} |
| \texttt{fMx} | \texttt{_FloatNx} for minimum \texttt{N > M} if supported, else \texttt{_FloatN} for minimum \texttt{N > M} |
| \texttt{dM} | \texttt{_DecimalMx} if supported, else \texttt{_DecimalN} for minimum \texttt{N > M} |
| \texttt{dMx} | \texttt{_DecimalN} for minimum \texttt{N > M} if supported, else \texttt{_DecimalN} for minimum \texttt{N > M} |

**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;
_Float32x complex f32xc;
_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>( \cosh f64x(f64xc) )</td>
</tr>
<tr>
<td>( \text{pow}(dc, f128) )</td>
<td>( \text{cpowf}128(dc, f128) )</td>
</tr>
<tr>
<td>( \text{fmax}(f64, d) )</td>
<td>( \text{fmaxf}64(f64, d) )</td>
</tr>
<tr>
<td>( \text{fmax}(d, f32x) )</td>
<td>( \text{fmax}(d, f32x) ), the function, if the set of values of _Float32x is a subset of (or equivalent to) the set of values of \text{double}, or ( \text{fmaxf}32x(d, f32x) ), if the set of values of \text{double} is a proper subset of the set of values of _Float32x, or undefined, if neither of the sets of values of \text{double} and _Float32x is a subset of the other (and the sets are not equivalent)</td>
</tr>
<tr>
<td>( \text{pow}(f32x, n) )</td>
<td>( \text{powf}32x(f32x, n) )</td>
</tr>
<tr>
<td>( \text{fsub}(d, ld) )</td>
<td>( \text{fsub}l )</td>
</tr>
<tr>
<td>( \text{f32add}(f64x, f64) )</td>
<td>( \text{f32addf}64 )</td>
</tr>
<tr>
<td>( \text{f32sqrt}(n) )</td>
<td>( \text{f32sqrt}f64 )</td>
</tr>
<tr>
<td>( \text{f32mul}(f128, f32x) )</td>
<td>( \text{f32mul}f128 ) if _Float128 is at least as wide as _Float32x, or ( \text{f32mul}f32x ) if _Float32x is wider than _Float128</td>
</tr>
<tr>
<td>( \text{f32fma}(f32x, n, f32x) )</td>
<td>( \text{f32fma}f64 ) if _Float64 is at least as wide as _Float32x, or ( \text{f32fma}f32x ) if _Float32x is wider than _Float64</td>
</tr>
<tr>
<td>( \text{ddiv}(ld, f128) )</td>
<td>undefined</td>
</tr>
<tr>
<td>( \text{f32fma}(f64, d, f64) )</td>
<td>undefined</td>
</tr>
<tr>
<td>( \text{fmul}(dc, d) )</td>
<td>undefined</td>
</tr>
<tr>
<td>( \text{f32add}(f32, f32) )</td>
<td>( \text{f32addf}32 )</td>
</tr>
<tr>
<td>( \text{f32sqrt}(f32) )</td>
<td>( \text{f32sqrt}f64 ) if _Float64 is supported, else ( \text{f32sqrt}f64 )</td>
</tr>
<tr>
<td>( \text{f64div}(f32x, f32x) )</td>
<td>( \text{f64divf}64x(f32x, f32x) ) if _Float64 is supported, else ( \text{f64div}f128 )</td>
</tr>
</tbody>
</table>