Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 3:
Interchange and extended types

Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel système — Extensions à virgule flottante pour C —

Partie 3: Types d'échange et étendus
Foreword....................................................................................................................................iv
Introduction.............................................................................................................................................vi
1 Scope.....................................................................................................................................................1
2 Conformance.............................................................................................................................................1
3 Normative references............................................................................................................................1
4 Terms and definitions ............................................................................................................................1
5 C standard conformance .......................................................................................................................2
  5.1 Freestanding implementations .......................................................................................................2
  5.2 Predefined macros ...........................................................................................................................2
  5.3 Standard headers .............................................................................................................................2
6 Types .....................................................................................................................................................8
7 Characteristics .....................................................................................................................................13
8 Conversions .........................................................................................................................................19
9 Constants ...........................................................................................................................................21
10 Expressions ......................................................................................................................................22
11 Non-arithmetic interchange formats ..................................................................................................23
12 Mathematics <math.h> .......................................................................................................................23
  12.1 Macros ..........................................................................................................................................25
  12.2 Floating-point environment .........................................................................................................29
  12.3 Functions ......................................................................................................................................31
  12.4 Encoding conversion functions ....................................................................................................42
13 Numeric conversion functions in <stdlib.h> ....................................................................................45
14 Complex arithmetic <complex.h> ....................................................................................................51
15 Type-generic macros <tgmath.h> .....................................................................................................54
Bibliography.........................................................................................................................................58
ISO/IEC TS 18661-3:2015(E)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO’s adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments, and system software interfaces.

ISO/IEC TS 18661 consists of the following parts, under the general title Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C:

— Part 1: Binary floating-point arithmetic
— Part 2: Decimal floating-point arithmetic
— Part 3: Interchange and extended types
— Part 4: Supplementary functions

The following part is under preparation:

— Part 5: Supplementary attributes


Introduction

Background

IEC 60559 floating-point standard

The IEEE 754-1985 standard for binary floating-point arithmetic was motivated by an expanding diversity in floating-point data representation and arithmetic, which made writing robust programs, debugging, and moving programs between systems exceedingly difficult. Now the great majority of systems provide data formats and arithmetic operations according to this standard. The IEC 60559:1989 international standard was equivalent to the IEEE 754-1985 standard. Its stated goals were the following:

1. Facilitate movement of existing programs from diverse computers to those that adhere to this standard.

2. Enhance the capabilities and safety available to programmers who, though not expert in numerical methods, may well be attempting to produce numerically sophisticated programs. However, we recognize that utility and safety are sometimes antagonists.

3. Encourage experts to develop and distribute robust and efficient numerical programs that are portable, by way of minor editing and recompilation, onto any computer that conforms to this standard and possesses adequate capacity. When restricted to a declared subset of the standard, these programs should produce identical results on all conforming systems.

4. Provide direct support for
   a. Execution-time diagnosis of anomalies
   b. Smoother handling of exceptions
   c. Interval arithmetic at a reasonable cost

5. Provide for development of
   a. Standard elementary functions such as exp and cos
   b. Very high precision (multiword) arithmetic
   c. Coupling of numerical and symbolic algebraic computation

6. Enable rather than preclude further refinements and extensions.

To these ends, the standard specified a floating-point model comprising the following:

- **formats** – for binary floating-point data, including representations for Not-a-Number (NaN) and signed infinities and zeros
- **operations** – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system; also specified conversions between floating-point formats and decimal character sequences, and a few auxiliary operations
- **context** – status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact) and controls for choosing different rounding methods
The ISO/IEC/IEEE 60559:2011 international standard is equivalent to the IEEE 754-2008 standard for floating-point arithmetic, which is a major revision to IEEE 754-1985.

The revised standard specifies more formats, including decimal as well as binary. It adds a 128-bit binary format to its basic formats. It defines extended formats for all of its basic formats. It specifies data interchange formats (which may or may not be arithmetic), including a 16-bit binary format and an unbounded tower of wider formats. To conform to the floating-point standard, an implementation must provide at least one of the basic formats, along with the required operations.

The revised standard specifies more operations. New requirements include – among others – arithmetic operations that round their result to a narrower format than the operands (with just one rounding), more conversions with integer types, more classifications and comparisons, and more operations for managing flags and modes. New recommendations include an extensive set of mathematical functions and seven reduction functions for sums and scaled products.

The revised standard places more emphasis on reproducible results, which is reflected in its standardization of more operations. For the most part, behaviors are completely specified. The standard requires conversions between floating-point formats and decimal character sequences to be correctly rounded for at least three more decimal digits than is required to distinguish all numbers in the widest supported binary format; it fully specifies conversions involving any number of decimal digits. It recommends that transcendental functions be correctly rounded.

The revised standard requires a way to specify a constant rounding direction for a static portion of code, with details left to programming language standards. This feature potentially allows rounding control without incurring the overhead of runtime access to a global (or thread) rounding mode.

Other features recommended by the revised standard include alternate methods for exception handling, controls for expression evaluation (allowing or disallowing various optimizations), support for fully reproducible results, and support for program debugging.

The revised standard, like its predecessor, defines its model of floating-point arithmetic in the abstract. It neither defines the way in which operations are expressed (which might vary depending on the computer language or other interface being used), nor does it define the concrete representation (specific layout in storage, or in a processor’s register, for example) of data or context, except that it does define specific encodings that are to be used for the exchange of floating-point data between different implementations that conform to the specification.

IEC 60559 does not include bindings of its floating-point model for particular programming languages. However, the revised standard does include guidance for programming language standards, in recognition of the fact that features of the floating-point standard, even if well supported in the hardware, are not available to users unless the programming language provides a commensurate level of support. The implementation’s combination of both hardware and software determines conformance to the floating-point standard.

C support for IEC 60559

The C standard specifies floating-point arithmetic using an abstract model. The representation of a floating-point number is specified in an abstract form where the constituent components (sign, exponent, significand) of the representation are defined but not the internals of these components. In particular, the exponent range, significand size, and the base (or radix) are implementation-defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of operations are also implementation-defined, for example in the area of handling of special numbers and in exceptions.
The reason for this approach is historical. At the time when C was first standardized, before the floating-point standard was established, there were various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would have made most of the existing implementations at the time not conforming.

Beginning with ISO/IEC 9899:1999 (C99), C has included an optional second level of specification for implementations supporting the floating-point standard. C99, in conditionally normative annex F, introduced nearly complete support for the IEC 60559:1989 standard for binary floating-point arithmetic. Also, C99’s informative annex G offered a specification of complex arithmetic that is compatible with IEC 60559:1989.


**Purpose**

The purpose of ISO/IEC TS 18661 is to provide a C language binding for ISO/IEC/IEEE 60559:2011, based on the C11 standard, that delivers the goals of ISO/IEC/IEEE 60559 to users and is feasible to implement. It is organized into five parts.

ISO/IEC TS 18661-1 provides changes to C11 that cover all the requirements, plus some basic recommendations, of ISO/IEC/IEEE 60559:2011 for binary floating-point arithmetic. C implementations intending to support ISO/IEC/IEEE 60559:2011 are expected to conform to conditionally normative annex F as enhanced by the changes in ISO/IEC TS 18661-1.


ISO/IEC TS 18661-3 (Interchange and extended types), ISO/IEC TS 18661-4 (Supplementary functions), and ISO/IEC TS 18661-5 (Supplementary attributes) cover recommended features of ISO/IEC/IEEE 60559:2011. C implementations intending to provide extensions for these features are expected to conform to the corresponding parts.

**Additional background on formats**

The revised floating-point arithmetic standard, ISO/IEC/IEEE 60559:2011, introduces a variety of new formats, both fixed and extendable. The new fixed formats include

- a 128-bit basic binary format (the 32 and 64 bit basic binary formats are carried over from ISO/IEC 60559:1989)
- 64 and 128 bit basic decimal formats
— interchange formats, whose precision and range are determined by the width \( k \), where

for binary, \( k = 16, 32, 64 \), and \( k \geq 128 \) and a multiple of 32, and

for decimal, \( k \geq 32 \) and a multiple of 32

— extended formats, for each basic format, with minimum range and precision specified

Thus IEC 60559 defines five basic formats — binary32, binary64, binary128, decimal64, and decimal128 — and five corresponding extended formats, each with somewhat more precision and range than the basic format it extends. IEC 60559 defines an unlimited number of interchange formats, which include the basic formats.

Interchange formats may or may not be supported as arithmetic formats. If not, they may be used for the interchange of floating-point data but not for arithmetic computation. IEC 60559 provides conversions between non-arithmetic interchange formats and arithmetic formats which can be used for computation.

Extended formats are intended for intermediate computation, not input or output data. The extra precision often allows the computation of extended results which when converted to a narrower output format differ from the ideal results by little more than a unit in the last place. Also, the extra range often avoids any intermediate overflow or underflow that might occur if the computation were done in the format of the data. The essential property of extended formats is their sufficient extra widths. Extended formats for any given basic format may vary among implementations.

Extendable formats, which provide user control over range and precision, are not covered in ISO/IEC TS 18661.

The 32 and 64 bit binary formats are supported in C by types `float` and `double`. If a C implementation defines the macro `__STDC_IEC_60559_BFP__` (see ISO/IEC TS 18661-1) signifying that it supports C Annex F for binary floating-point arithmetic, then its `float` and `double` formats must be IEC 60559 binary32 and binary64.

ISO/IEC TS 18661-2 defines types `__Decimal32`, `__Decimal64`, and `__Decimal128` with IEC 60559 formats decimal32, decimal64, and decimal128. Although IEC 60559 does not require arithmetic support (other than conversions) for its decimal32 interchange format, ISO/IEC TS 18661-2 has full arithmetic and library support for `__Decimal32`, just like for `__Decimal64` and `__Decimal128`.

The C Standard provides just three standard floating types (`float`, `double`, and `long double`) that are required of all implementations. C Annex F for binary floating-point arithmetic requires the standard floating types to be binary. The `long double` type must be at least as wide as `double`, but C does not further specify details of its format, even in Annex F.

ISO/IEC TS 18661-3, this document, provides nomenclatures for types with IEC 60559 arithmetic interchange formats and extended formats. The nomenclatures allow portable use of the formats as envisioned in IEC 60559. This document covers these aspects of the types:

— names

— characteristics

— conversions
This specification includes interchange and extended nomenclatures for formats that, in some cases, already have C nomenclatures. For example, types with the IEC 60559 double format may include `double, _Float64` (the type for the binary64 interchange format), and maybe `_Float32x` (the type for the binary32-extended format). This redundancy is intended to support the different programming models appropriate for the types with arithmetic interchange formats and extended formats and C standard floating types.

This document also supports the IEC 60559 non-arithmetic interchange formats with functions that convert among encodings and between encodings and character sequences, for all interchange formats.
Information technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C —

Part 3:
Interchange and extended types

1 Scope

This part of ISO/IEC TS 18661 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.

2 Conformance

An implementation conforms to this part of ISO/IEC TS 18661 if

a) it meets the requirements for a conforming implementation of C11 with all the changes to C11 specified in parts 1-3 of ISO/IEC TS 18661;

b) it conforms to ISO/IEC TS 18661-1 or ISO/IEC TS 18661-2 (or both); and

c) it defines __STDC_IEC_60559_TYPES__ to 201506L.

3 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.


ISO/IEC TS 18661-2:2015, Information technology — Programming languages, their environments and system software interfaces — Floating-point extensions for C — Part 2: Decimal floating-point arithmetic

4 Terms and definitions

4.1

C11


5  C standard conformance

5.1 Freestanding implementations

The specification in C11 + TS18661-1 + TS18661-2 allows freestanding implementations to conform to this part of ISO/IEC TS 18661.

5.2 Predefined macros

Change to C11 + TS18661-1 + TS18661-2:

In 6.10.8.3#1, add:

```c
__STDC_IEC_60559_TYPES__ The integer constant 201506L, intended to indicate support of interchange and extended floating types according to IEC 60559.
```

5.3 Standard headers

The new identifiers added to C11 library headers by this part of ISO/IEC TS 18661 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 changes to C11 + TS18661-1 + TS18661-2 list these identifiers in each applicable library subclause.

**Changes to C11 + TS18661-1 + TS18661-2:**

After 5.2.4.2.2#6b, insert the paragraph:

[6c] 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:

```
FLT_N_MANT_DIG        FLT_N_MIN_10_EXP     FLT_N_EPSILON
FLT_N_DECIMAL_DIG     FLT_N_MAX_EXP      FLT_N_MIN
FLT_N_DIG             FLT_N_MAX_10_EXP    FLT_N_TRUE_MIN
FLT_N_MIN_EXP         FLT_N_MAX
```

for supported types _FloatNx:

```
FLT_NX_MANT_DIG       FLT_NX_MIN_10_EXP   FLT_NX_EPSILON
FLT_NX_DECIMAL_DIG    FLT_NX_MAX_EXP     FLT_NX_MIN
FLT_NX_DIG            FLT_NX_MAX_10_EXP   FLT_NX_TRUE_MIN
FLT_NX_MIN_EXP        FLT_NX_MAX
```
for supported types \texttt{\_DecimalN}, where $N \neq 32, 64, \text{ and } 128$:

\begin{align*}
\text{DECN\_MANT\_DIG} & \quad \text{DECN\_MAX} & \quad \text{DECN\_TRUE\_MIN} \\
\text{DECN\_MIN\_EXP} & \quad \text{DECN\_EPSILON} & \quad \text{DECN\_MIN} \\
\text{DECN\_MAX\_EXP} & \quad \text{DECN\_MIN} & \quad \text{DECN\_MIN}
\end{align*}

for supported types \texttt{\_DecimalNx}:

\begin{align*}
\text{DECNX\_MANT\_DIG} & \quad \text{DECNX\_MAX} & \quad \text{DECNX\_TRUE\_MIN} \\
\text{DECNX\_MIN\_EXP} & \quad \text{DECNX\_EPSILON} & \quad \text{DECNX\_MIN} \\
\text{DECNX\_MAX\_EXP} & \quad \text{DECNX\_MIN} & \quad \text{DECNX\_MIN}
\end{align*}

After 7.3\#2, insert the paragraph:

[2a] The following identifiers are declared or defined only if \texttt{\_STDC\_WANT\_IEC\_60559\_TYPES\_EXT} is defined as a macro at the point in the source file where \texttt{\langle complex.h\rangle} is first included:

for supported types \texttt{\_FloatN}:

\begin{align*}
\text{cacosfN} & \quad \text{catanhfN} & \quad \text{csqrtfN} \\
\text{casinfN} & \quad \text{ccoshfN} & \quad \text{cargfN} \\
\text{catanfN} & \quad \text{csinhfN} & \quad \text{cimagfN} \\
\text{ccosfN} & \quad \text{ctanhfN} & \quad \text{CMPLXFN} \\
\text{csinfN} & \quad \text{cexpfN} & \quad \text{conjfN} \\
\text{ctanfN} & \quad \text{clogfN} & \quad \text{cprojfN} \\
\text{cacosfN} & \quad \text{cabsfN} & \quad \text{crealfN} \\
\text{casinhfN} & \quad \text{cpowfN} & \quad \text{CREALFN}
\end{align*}

for supported types \texttt{\_FloatNx}:

\begin{align*}
\text{cacosfNx} & \quad \text{catanhfNx} & \quad \text{csqrtfNx} \\
\text{casinfNx} & \quad \text{ccoshfNx} & \quad \text{cargfNx} \\
\text{catanfNx} & \quad \text{csinhfNx} & \quad \text{cimagfNx} \\
\text{ccosfNx} & \quad \text{ctanhfNx} & \quad \text{CMPLXFN} \\
\text{csinfNx} & \quad \text{cexpfNx} & \quad \text{conjfNx} \\
\text{ctanfNx} & \quad \text{clogfNx} & \quad \text{cprojfNx} \\
\text{cacosfNx} & \quad \text{cabsfNx} & \quad \text{crealfNx} \\
\text{casinhfNx} & \quad \text{cpowfNx} & \quad \text{CREALFN}
\end{align*}

After 7.12\#1c, insert the paragraph:

[1d] The following identifiers are defined or 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 math.h\rangle} is first included:

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

for supported types \texttt{\_FloatN}:

\begin{align*}
\texttt{\_FloatN\_t} & \quad \text{log1pfN} & \quad \text{fromfpfN} \\
\texttt{\_HUGE\_VAL\_FN} & \quad \text{log2fN} & \quad \text{ufromfpfN}
\end{align*}
ISO/IEC TS 18661-3:2015(E)

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>logbfN</td>
<td>modffN</td>
<td>fromfpfxN</td>
</tr>
<tr>
<td>FP_FAST_FMAF</td>
<td>scalbnfN</td>
<td>ufromfpfxN</td>
</tr>
<tr>
<td>acosfN</td>
<td>scalblnfN</td>
<td>fmodfN</td>
</tr>
<tr>
<td>asinfN</td>
<td>cbrtfN</td>
<td>remainderfN</td>
</tr>
<tr>
<td>atanfN</td>
<td>fabsfN</td>
<td>remquofN</td>
</tr>
<tr>
<td>atan2fN</td>
<td>hypotfN</td>
<td>copysignfN</td>
</tr>
<tr>
<td>cosfN</td>
<td>powfN</td>
<td>nanfN</td>
</tr>
<tr>
<td>sinfN</td>
<td>sqrtfN</td>
<td>nextafterfN</td>
</tr>
<tr>
<td>tanfN</td>
<td>erfN</td>
<td>nextupfN</td>
</tr>
<tr>
<td>acoshfN</td>
<td>erffN</td>
<td>nextdownfN</td>
</tr>
<tr>
<td>asinhfN</td>
<td>lgammafN</td>
<td>canonicalizefN</td>
</tr>
<tr>
<td>atanhfN</td>
<td>tgammafN</td>
<td>encodefN</td>
</tr>
<tr>
<td>coshfN</td>
<td>scalblnfN</td>
<td>decodefN</td>
</tr>
<tr>
<td>sinhN</td>
<td>cbrtfN</td>
<td>fdimfN</td>
</tr>
<tr>
<td>tanhfN</td>
<td>fmodfN</td>
<td>fmaxfN</td>
</tr>
<tr>
<td>expN</td>
<td>roundfN</td>
<td>fminfN</td>
</tr>
<tr>
<td>exp2fN</td>
<td>nearbyintfN</td>
<td>fmaxmagfN</td>
</tr>
<tr>
<td>expmlfN</td>
<td>rintfN</td>
<td>fminmagfN</td>
</tr>
<tr>
<td>frexpN</td>
<td>lrintfN</td>
<td>fmafN</td>
</tr>
<tr>
<td>ilogbfN</td>
<td>llrintfN</td>
<td>totalorderfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>roundfN</td>
<td>totalordermagfN</td>
</tr>
<tr>
<td>logbfN</td>
<td>truncfN</td>
<td>getpayloadfN</td>
</tr>
<tr>
<td>log10fN</td>
<td>roundevenfN</td>
<td>setpayloadfN</td>
</tr>
<tr>
<td>logfN</td>
<td></td>
<td>setpayloadsigfN</td>
</tr>
</tbody>
</table>

for supported types _FloatNx:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUGE_VAL_FNX</td>
<td>logbfNx</td>
<td>fromfpfxN</td>
</tr>
<tr>
<td>SNANFNx</td>
<td>modffN</td>
<td>ufromfpfxN</td>
</tr>
<tr>
<td>FP_FAST_FMAFNx</td>
<td>scalbnfNx</td>
<td>fmodfN</td>
</tr>
<tr>
<td>acosfNx</td>
<td>scalblnfNx</td>
<td>remainderfN</td>
</tr>
<tr>
<td>asinfNx</td>
<td>cbrtfNx</td>
<td>remquofN</td>
</tr>
<tr>
<td>atanfNx</td>
<td>fabsfNx</td>
<td>copysignfN</td>
</tr>
<tr>
<td>atan2fNx</td>
<td>hypotfNx</td>
<td>nanfN</td>
</tr>
<tr>
<td>cosfNx</td>
<td>powfNx</td>
<td>nextafterfN</td>
</tr>
<tr>
<td>sinfNx</td>
<td>sqrtfNx</td>
<td>nextupfN</td>
</tr>
<tr>
<td>tanfNx</td>
<td>erfN</td>
<td>nextdownfN</td>
</tr>
<tr>
<td>acoshfNx</td>
<td>erffN</td>
<td>canonicalizefN</td>
</tr>
<tr>
<td>asinhfNx</td>
<td>lgammafN</td>
<td>encodefN</td>
</tr>
<tr>
<td>atanhfNx</td>
<td>tgammafN</td>
<td>decodefN</td>
</tr>
<tr>
<td>coshfNx</td>
<td>scalblnfN</td>
<td>fdimfN</td>
</tr>
<tr>
<td>sinhfNx</td>
<td>cbrtfN</td>
<td>fmaxfN</td>
</tr>
<tr>
<td>tanhfNx</td>
<td>fmodfN</td>
<td>fminfN</td>
</tr>
<tr>
<td>expfN</td>
<td>roundfN</td>
<td>fmaxmagfN</td>
</tr>
<tr>
<td>exp2fN</td>
<td>nearbyintfN</td>
<td>fminmagfN</td>
</tr>
<tr>
<td>expmlfN</td>
<td>rintfN</td>
<td>fmafN</td>
</tr>
<tr>
<td>frexpfN</td>
<td>lrintfN</td>
<td>totalorderfN</td>
</tr>
<tr>
<td>ilogbfN</td>
<td>llrintfN</td>
<td>totalordermagfN</td>
</tr>
<tr>
<td>ldexpfN</td>
<td>roundfN</td>
<td></td>
</tr>
<tr>
<td>logfN</td>
<td>lroundfN</td>
<td></td>
</tr>
<tr>
<td>log10fN</td>
<td>lroundfN</td>
<td></td>
</tr>
</tbody>
</table>
for supported types _FloatM and _FloatN where M < N:

\[
\begin{align*}
\text{FP\_FAST\_FMADDFN} & \quad \text{FP\_FAST\_FMFAFN} & \quad fMmulfN \\
\text{FP\_FAST\_FMSUBFN} & \quad \text{FP\_FAST\_FMSQRTFN} & \quad fMdivfN \\
\text{FP\_FAST\_FMULFN} & \quad fMaddfN & \quad fMmafnN \\
\text{FP\_FAST\_FMDIVFN} & \quad fMsubfN & \quad fMsqrtfN
\end{align*}
\]

for supported types _FloatM and _FloatN where M ≤ N:

\[
\begin{align*}
\text{FP\_FAST\_FMADDFN} & \quad \text{FP\_FAST\_FMFAFN} & \quad fMmulfN \\
\text{FP\_FAST\_FMSUBFN} & \quad \text{FP\_FAST\_FMSQRTFN} & \quad fMdivfN \\
\text{FP\_FAST\_FMULFN} & \quad fMaddfN & \quad fMmafnN \\
\text{FP\_FAST\_FMDIVFN} & \quad fMsubfN & \quad fMsqrtfN
\end{align*}
\]

for supported types _FloatMx and _FloatN where M < N:

\[
\begin{align*}
\text{FP\_FAST\_FMXADDFN} & \quad \text{FP\_FAST\_FMXFAFN} & \quad fMxmulfN \\
\text{FP\_FAST\_FMXSUBFN} & \quad \text{FP\_FAST\_FMXSQRTFN} & \quad fMxdivfN \\
\text{FP\_FAST\_FMXULFN} & \quad fMxaddfN & \quad fMxmafnN \\
\text{FP\_FAST\_FMXDVFN} & \quad fMxsubfN & \quad fMxsqrtfN
\end{align*}
\]

for supported types _FloatMx and _FloatN where M < N:

\[
\begin{align*}
\text{FP\_FAST\_FMXADDFN} & \quad \text{FP\_FAST\_FMXFAFN} & \quad fMxmulfN \\
\text{FP\_FAST\_FMXSUBFN} & \quad \text{FP\_FAST\_FMXSQRTFN} & \quad fMxdivfN \\
\text{FP\_FAST\_FMXULFN} & \quad fMxaddfN & \quad fMxmafnN \\
\text{FP\_FAST\_FMXDVFN} & \quad fMxsubfN & \quad fMxsqrtfN
\end{align*}
\]

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

\[
fMencfN
\]

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

\[
\begin{align*}
\text{logbdN} & \quad \text{fmoddN} & \quad \text{remainderdN} \\
\text{modfdN} & \quad \text{copysigndN} & \quad \text{nandN} \\
\text{scalbndN} & \quad \text{nextafterdN} & \quad \text{nextupdN} \\
\text{scalblndN} & \quad \text{canonicalizedN} & \quad \text{quantizedN} \\
\text{cbrtdN} & \quad \text{samequantumdN} & \quad \text{quantumdN} \\
\text{fabsdN} & \quad \text{lgammaN} & \quad \text{llquantexpdN} \\
\text{hypotdN} & \quad \text{llquantN} & \quad \text{encodedcN}
\end{align*}
\]
atanhd\(N\)  &  ceild\(N\)  &  decoded\(cd\)N  
coshd\(N\)  &  floord\(N\)  &  encodebind\(N\)  
sinhd\(N\)  &  nearbyintd\(N\)  &  decodebind\(N\)  
tanhd\(N\)  &  rintd\(N\)  &  fdimd\(N\)  
expd\(N\)  &  lrintd\(N\)  &  fmaxd\(N\)  
exp2d\(N\)  &  llrintd\(N\)  &  fminm\(N\)magd\(N\)  
expmld\(N\)  &  roundd\(N\)  &  fmaxm\(N\)agd\(N\)  
frexp\(d\)\(N\)  &  lroundd\(N\)  &  fminm\(N\)agd\(N\)  
ilogbd\(N\)  &  lroundd\(N\)  &  fmad\(N\)  
ilogbd\(N\)  &  truncd\(N\)  &  totalorderd\(N\)  
lde\(xp\)d\(N\)  &  rounde\(v\)nd\(N\)  &  totalorderm\(ag\)d\(N\)  
logd\(N\)  &  fromf\(p\)d\(N\)  &  getpayload\(d\)\(N\)  
log10d\(N\)  &  ufromf\(p\)d\(N\)  &  setpayload\(d\)\(N\)  
log1pd\(N\)  &  fromfp\(x\)d\(N\)  &  setpayload\(s\)i\(g\)d\(N\)  
log2d\(N\)  &  ufromfp\(x\)d\(N\)  &  

for supported types **\_Decimal\(\_\)\(x\):**

| **HUGE\_VAL\_D\(\_\)\(N\)x** | **log2d\(N\)x** | **ufromfp\(x\)d\(N\)x** |
| **SNAND\(\_\)\(N\)x** | **logbd\(N\)x** | **fromfp\(x\)d\(N\)x** |
| **FP\_FAST\_FMAD\(\_\)\(N\)x** | **modfd\(N\)x** | **ufromfp\(x\)d\(N\)x** |
| **acosd\(N\)x** | **scalbnd\(N\)x** | **fmodd\(N\)x** |
| **asind\(N\)x** | **scalblnd\(N\)x** | **remainderd\(N\)x** |
| **atand\(N\)x** | **cbrtd\(N\)x** | **copysign\(d\)N\x** |
| **atan2d\(N\)x** | **fabsd\(N\)x** | **nand\(N\)x** |
| **cosd\(N\)x** | **hypotd\(N\)x** | **nextaft\(e\)rd\(N\)x** |
| **sind\(N\)x** | **powd\(N\)x** | **next\(t\)up\(d\)N\x** |
| **tand\(N\)x** | **sqrtd\(N\)x** | **next\(d\)ow\(n\)d\(N\)x** |
| **acoshd\(N\)x** | **erfd\(N\)x** | **canon\(o\)nalized\(N\)x** |
| **asinhd\(N\)x** | **erfc\(d\)N\x** | **quantized\(N\)x** |
| **atanhd\(N\)x** | **lgam\(m\)ad\(N\)x** | **samequantum\(d\)N\x** |
| **coshd\(N\)x** | **tgam\(m\)ad\(N\)x** | **quantum\(d\)N\x** |
| **sinhd\(N\)x** | **ceild\(N\)x** | **llquantexp\(d\)N\x** |
| **tanhd\(N\)x** | **floor\(d\)N\x** | **fdimd\(N\)x** |
| **expd\(N\)x** | **nearbyintd\(N\)x** | **fmaxd\(N\)x** |
| **exp2d\(N\)x** | **rintd\(N\)x** | **fminm\(N\)agd\(N\)x** |
| **expmld\(N\)x** | **lrintd\(N\)x** | **fmaxm\(N\)agd\(N\)x** |
| **frexp\(d\)\(N\)x** | **llrintd\(N\)x** | **fminm\(N\)agd\(N\)x** |
| **ilogbd\(N\)x** | **roundd\(N\)x** | **fmad\(N\)x** |
| **llogbd\(N\)x** | **lroundd\(N\)x** | **totalorderd\(N\)x** |
| **ldexp\(d\)\(N\)x** | **llroundd\(N\)x** | **totalorderm\(ag\)d\(N\)x** |
| **logd\(N\)x** | **truncd\(N\)x** | **getpayload\(d\)\(N\)x** |
| **log10d\(N\)x** | **rounde\(v\)nd\(N\)x** | **setpayload\(d\)\(N\)x** |
| **log1pd\(N\)x** | **fromfp\(d\)N\x** | **setpayload\(i\)g\(d\)N\x** |
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:

\[
\begin{align*}
\text{FP\_FAST\_D\ histoADDN} & \quad \text{FP\_FAST\_D\ histoFMADN} & \quad \text{dMmuldN} \\
\text{FP\_FAST\_D\ histoSUBDN} & \quad \text{FP\_FAST\_D\ histoSQRTDN} & \quad \text{dMdivdN} \\
\text{FP\_FAST\_D\ histoMULDN} & \quad \text{dMadddN} & \quad \text{dMfmadN} \\
\text{FP\_FAST\_D\ histoDIVDN} & \quad \text{dMsubdN} & \quad \text{dMsqrtdN}
\end{align*}
\]

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

\[
\begin{align*}
\text{FP\_FAST\_D\ histoADDN} & \quad \text{FP\_FAST\_D\ histoFMADN} & \quad \text{dMmuldN} \\
\text{FP\_FAST\_D\ histoSUBDN} & \quad \text{FP\_FAST\_D\ histoSQRTDN} & \quad \text{dMdivdN} \\
\text{FP\_FAST\_D\ histoMULDN} & \quad \text{dMadddN} & \quad \text{dMfmadN} \\
\text{FP\_FAST\_D\ histoDIVDN} & \quad \text{dMsubdN} & \quad \text{dMsqrtdN}
\end{align*}
\]

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

\[
\begin{align*}
\text{FP\_FAST\_D\ histoADDN} & \quad \text{FP\_FAST\_D\ histoFMADN} & \quad \text{dMmuldN} \\
\text{FP\_FAST\_D\ histoSUBDN} & \quad \text{FP\_FAST\_D\ histoSQRTDN} & \quad \text{dMdivdN} \\
\text{FP\_FAST\_D\ histoMULDN} & \quad \text{dMadddN} & \quad \text{dMfmadN} \\
\text{FP\_FAST\_D\ histoDIVDN} & \quad \text{dMsubdN} & \quad \text{dMsqrtdN}
\end{align*}
\]

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

\[
\begin{align*}
\text{FP\_FAST\_D\ histoADDN} & \quad \text{FP\_FAST\_D\ histoFMADN} & \quad \text{dMmuldN} \\
\text{FP\_FAST\_D\ histoSUBDN} & \quad \text{FP\_FAST\_D\ histoSQRTDN} & \quad \text{dMdivdN} \\
\text{FP\_FAST\_D\ histoMULDN} & \quad \text{dMadddN} & \quad \text{dMfmadN} \\
\text{FP\_FAST\_D\ histoDIVDN} & \quad \text{dMsubdN} & \quad \text{dMsqrtdN}
\end{align*}
\]

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

\[
\text{dMencdec}dN \quad \text{dMencbind}N
\]

After 7.22#1b, insert the paragraph:

[1c] 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{\_Float}N:

\[
\begin{align*}
\text{strfromf}N \quad \text{strtof}N \\
\end{align*}
\]

for supported types \texttt{\_Float}Nx:

\[
\begin{align*}
\text{strfromf}Nx \quad \text{strtof}Nx \\
\end{align*}
\]

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

\[
\begin{align*}
\text{strfromd}N \quad \text{strtod}N \\
\end{align*}
\]
for supported types _DecimalNx:

\texttt{strfromdNx} \hspace{1cm} \texttt{strtod Nx}

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

\texttt{strfromencfN} \hspace{1cm} \texttt{strtoencfN}

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

\texttt{strfromencdec dN} \hspace{1cm} \texttt{strtoencdec dN}
\texttt{strfromencbindN} \hspace{1cm} \texttt{strtoencbindN}

6 Types

This clause specifies changes to C11 + TS18661-1 + TS18661-2 to include types that support IEC 60559 arithmetic formats:

\_\texttt{FloatN} for binary interchange formats

\_\texttt{DecimalN} for decimal interchange formats

\_\texttt{FloatNx} for binary extended formats

\_\texttt{DecimalNx} for decimal extended formats

The encoding conversion functions (12.4) and numeric conversion functions for encodings (13) support the non-arithmetic interchange formats specified in IEC 60559.

ISO/IEC TS 18661-2 defined \textit{standard floating types} as a collective name for the types \texttt{float}, \texttt{double}, and \texttt{long double} and it defined \textit{decimal floating types} as a collective name for the types \_\texttt{Decimal32}, \_\texttt{Decimal64}, and \_\texttt{Decimal128}. This part of ISO/IEC TS 18661 extends the definition of decimal floating types and defines \textit{binary floating types} to be collective names for types for all the appropriate IEC 60559 arithmetic formats. Thus real floating types are classified as follows:

\texttt{standard floating types:}
\texttt{float}
\texttt{double}
\texttt{long double}

\texttt{binary floating types:}
\_\texttt{FloatN}
\_\texttt{FloatNx}

\texttt{decimal floating types:}
\_\texttt{DecimalN}
\_\texttt{DecimalNx}

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).
Changes to C11 + TS18661-1 + TS18661-2:

Replace 6.2.5#10a-10b:

[10a] There are three *decimal floating types*, designated as \( \texttt{Decimal32} \), \( \texttt{Decimal64} \), and \( \texttt{Decimal128} \). Respectively, they have the IEC 60559 formats: decimal32, decimal64, and decimal128. Decimal floating types are real floating types.

[10b] Together, the standard floating types and the decimal floating types comprise the *real floating types*.

with:

[10a] 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>( bias, E-e )</th>
<th>15</th>
<th>127</th>
<th>1023</th>
<th>16383</th>
<th>( emax )</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 \( emax = 262143 \).
### Decimal interchange format parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>decimal32</th>
<th>decimal64</th>
<th>decimal128</th>
<th>decimalN (N ≥ 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>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$bias, E-e$</td>
<td>101</td>
</tr>
<tr>
<td>sign bit</td>
<td>1</td>
</tr>
<tr>
<td>$W+5$, combination field width in bits</td>
<td>11</td>
</tr>
<tr>
<td>$t$, trailing significand field width in bits</td>
<td>20</td>
</tr>
<tr>
<td>$N$, storage width in bits</td>
<td>32</td>
</tr>
</tbody>
</table>

For example, decimal256 would have $p = 70$ and $emax = 1572864$.

[10b] Types designated

- **_FloatN**, where $N$ is 16, 32, 64, or ≥ 128 and a multiple of 32

and types designated

- **_DecimalN**, where $N$ ≥ 32 and a multiple of 32

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

[10c] An implementation that defines **__STDC_IEC_60559_BFP__** and **__STDC_IEC_60559_TYPES__** shall provide **_Float32** and **_Float64** as interchange floating types with the same representation and alignment requirements as **float** and **double**, respectively. If the implementation’s **long double** type supports an IEC 60559 interchange format of width $N > 64$, then the implementation shall also provide the type **_FloatN** as an interchange floating type with the same representation and alignment requirements as **long double**. The implementation may provide other binary interchange floating types; the set of such types supported is implementation-defined.

[10d] 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.

[10e] Note that 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 (7.12.11.1c), string-to-encoding functions (7.22.1.3c), and string-from-encoding functions (7.22.1.3d). An implementation that defines **__STDC_IEC_60559_TYPES__** shall support the IEC 60559 binary16 format, at least as a non-arithmetic interchange format; the set of non-arithmetic interchange formats supported is implementation-defined.
[10f] 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:

**Extended format parameters for floating-point numbers**

<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>emax ≥</td>
<td>1023</td>
<td>16383</td>
<td>65535</td>
<td>6144</td>
<td>24576</td>
</tr>
</tbody>
</table>

[10g] Types designated _Float32x, _Float64x, _Float128x, _Decimal164x, and _Decimal128x support the corresponding IEC 60559 extended formats and are collectively called the extended floating types. Extended floating types are not compatible with any other types. 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: _Decimal164x, 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.

[10h] The standard floating types, interchange floating types, and extended floating types are collectively called the real floating types.

[10i] 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. Thus the binary floating types and the decimal floating types are real floating types.

The footnote reference above in new paragraph #10d is to the footnote referred to in removed paragraph #10a.

Replace 6.2.5#11:

[11] There are three complex types, designated as float _Complex, double _Complex, and long double _Complex; (Complex types are a conditional feature that implementations need not support; see 6.10.8.3.) The real floating and complex types are collectively called the floating types.

with:

[11] For the standard real types float, double, and long double, the interchange floating types _FloatN, and the extended floating types _FloatNx, there are complex types designated respectively as float _Complex, double _Complex, long double _Complex, _FloatN _Complex, and _FloatNx _Complex; (Complex types are a conditional feature that implementations need not support; see 6.10.8.3.) The real floating and complex types are collectively called the floating types.
In the list of keywords in 6.4.1, replace:

```plaintext
.Decimal32
.Decimal64
.Decimal128
```

with:

```plaintext
_Float\(N\), where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
_Float32x
_Float64x
_Float128x
_Decimal\(N\), where \(N\) \(\geq 32\) and a multiple of 32
_Decimal64x
_Decimal128x
```

In the list of type specifiers in 6.7.2, replace:

```plaintext
.Decimal32
.Decimal64
.Decimal128
```

with:

```plaintext
_Float\(N\), where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
_Float32x
_Float64x
_Float128x
_Decimal\(N\), where \(N\) \(\geq 32\) and a multiple of 32
_Decimal64x
_Decimal128x
```

In the list of constraints in 6.7.2#2, replace:

```plaintext
—_Decimal32
—_Decimal64
—_Decimal128
```

with:

```plaintext
—_Float\(N\), where \(N\) is 16, 32, 64, or \(\geq 128\) and a multiple of 32
—_Float32x
—_Float64x
—_Float128x
—_Decimal\(N\), where \(N\) \(\geq 32\) and a multiple of 32
—_Decimal64x
```
— _Decimal128x
— _FloatN _Complex, where N is 16, 32, 64, or ≥ 128 and a multiple of 32
— _Float32x _Complex
— _Float64x _Complex
— _Float128x _Complex

Replace 6.7.2#3a:

[3a] The type specifiers _Decimal32, _Decimal64, and _Decimal128 shall not be used if the implementation does not support decimal floating types (see 6.10.8.3).

with:

[3a] The type specifiers _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, and _Decimal128x shall not be used if the implementation does not support the corresponding types (see 6.10.8.3).

Replace 6.5#8a:

[8a] Operators involving decimal floating types are evaluated according to the semantics of IEC 60559, including production of results with the preferred quantum exponent as specified in IEC 60559.

with:

[8a] 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 5.2.4.2.2b).

Replace G.2#2:

[2] There are three imaginary types, designated as float _Imaginary, double _Imaginary, and long double _Imaginary. The imaginary types (along with the real floating and complex types) are floating types.

with:

[2] For the standard floating types float, double, and long double, the interchange floating types _FloatN, and the extended floating types _FloatNx, there are imaginary types designated respectively as float _Imaginary, double _Imaginary, long double _Imaginary, _FloatN _Imaginary, and _FloatNx _Imaginary. The imaginary types (along with the real floating and complex types) are floating types.

7 Characteristics

This clause specifies new <float.h> macros, analogous to the macros for standard floating types, that characterize the interchange and extended floating types. Some specification for decimal floating
types introduced in ISO/IEC TS 18661-2 is subsumed under the general specification for interchange floating types.

Changes to C11 + TS18661-1 + TS18661-2:

Renumber and rename 5.2.4.2.2a:

5.2.4.2.2a Characteristics of decimal floating types in <float.h>

to:

5.2.4.2.2b Alternate model for decimal floating-point numbers

and remove paragraphs 1-3:

[1] This subclause specifies macros in <float.h> that provide characteristics of decimal floating types in terms of the model presented in 5.2.4.2.2. The prefixes DEC32_, DEC64_, and DEC128_ denote the types _Decimal32, _Decimal64, and _Decimal128 respectively.

[2] DEC_EVAL_METHOD is the decimal floating-point analogue of FLT_EVAL_METHOD (5.2.4.2.2). Its implementation-defined value characterizes the use of evaluation formats for decimal floating types:

-1 indeterminable;

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

1 evaluate operations and constants of type _Decimal32 and _Decimal64 to the range and precision of the _Decimal64 type, evaluate _Decimal128 operations and constants to the range and precision of the _Decimal128 type;

2 evaluate all operations and constants to the range and precision of the _Decimal128 type.

[3] 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, $b(=10)$

For the standard floating types, this value is implementation-defined and is specified by the macro FLT_RADIX. For the decimal floating types there is no corresponding macro, since the value 10 is an inherent property of the types. Wherever FLT_RADIX appears in a description of a function that has versions that operate on decimal floating types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than FLT_RADIX.

— number of digits in the coefficient

| DEC32_MANT_DIG | 7 |
| DEC64_MANT_DIG | 16 |
| DEC128_MANT_DIG | 34 |
— minimum exponent

DEC32_MIN_EXP  -94
DEC64_MIN_EXP  -382
DEC128_MIN_EXP -6142

— maximum exponent

DEC32_MAX_EXP  97
DEC64_MAX_EXP  385
DEC128_MAX_EXP 6145

— maximum representable finite decimal floating-point number (there are 6, 15 and 33 9's after the decimal points respectively)

DEC32_MAX 9.999999E96DF
DEC64_MAX 9.999999999999999E384DD
DEC128_MAX 9.999999999999999999999999999999999E6144DL

— the difference between 1 and the least value greater than 1 that is representable in the given floating type

DEC32_EPSILON  1E-6DF
DEC64_EPSILON  1E-15DD
DEC128_EPSILON 1E-33DL

— minimum normalized positive decimal floating-point number

DEC32_MIN  1E-95DF
DEC64_MIN  1E-383DD
DEC128_MIN 1E-6143DL

— minimum positive subnormal decimal floating-point number

DEC32_TRUE_MIN 0.0000001E-95DF
DEC64_TRUE_MIN 0.0000000000000001E-383DD
DEC128_TRUE_MIN 0.0000000000000000000001E-6143DL

After 5.2.4.2.2, insert:

5.2.4.2.2a Characteristics of interchange and extended floating types in <float.h>

[1] This subclause specifies macros in <float.h> that provide characteristics of interchange floating types and extended floating types in terms of the model presented in 5.2.4.2.2. The prefix FLT_N_ indicates a binary interchange floating type of width N. The prefix FLT_NX_ indicates a binary extended floating type that extends a basic format of width N. The prefix DEC_N_ indicates a decimal interchange floating type of width N. The prefix DEC_NX_ indicates a decimal extended floating type that extends a basic format of width N. The type parameters p, e_{max}, and e_{min} for extended floating types are for the extended floating type itself, not for the basic format that it extends. For each interchange or extended floating type that the implementation provides, <float.h> shall define the associated macros in the following lists. Conversely, for each such type that the implementation does not provide, <float.h> shall not define the associated macros in the following lists.
[2] If `FLT_RADIX` is 2, the value of the macro `FLT_EVAL_METHOD` (5.2.4.2.2) characterizes the use of evaluation formats for standard floating types and for binary 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`\textsuperscript{\textit{N}} is a supported interchange floating type

- evaluate operations and constants, whose semantic type has at most the range and precision of the `_Float`\textsuperscript{\textit{N}} type, to the range and precision of the `_Float`\textsuperscript{\textit{N}} type; evaluate all other operations and constants to the range and precision of the semantic type;

`N + 1`, where `_Float`\textsuperscript{\textit{N}x} is a supported extended floating type

- evaluate operations and constants, whose semantic type has at most the range and precision of the `_Float`\textsuperscript{\textit{N}x} type, to the range and precision of the `_Float`\textsuperscript{\textit{N}x} 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` characterizes the use of evaluation formats (see analogous `FLT_EVAL_METHOD` in 5.2.4.2.2) 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; 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;
\( N \), where \(_\text{Decimal}N\) is a supported interchange floating type
evaluate operations and constants, whose semantic type has at most the range and
precision of the \(_\text{Decimal}N\) type, to the range and precision of the \(_\text{Decimal}N\)
type; evaluate all other operations and constants to the range and precision of the
semantic type;

\( N + 1 \), where \(_\text{Decimal}Nx\) is a supported extended floating type
evaluate operations and constants, whose semantic type has at most the range and
precision of the \(_\text{Decimal}Nx\) type, to the range and precision of the \(_\text{Decimal}Nx\)
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)

For the standard floating types, this value is implementation-defined and is specified by the
macro \texttt{FLT\_RADIX}. For the interchange and extended floating types there is no
Corresponding macro, since the radix is an inherent property of the types.

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

\[
\text{FLT}_N\_MANT\_DIG \\
\text{FLT}_N\times\_MANT\_DIG
\]

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

\[
\text{DEC}_N\_MANT\_DIG \\
\text{DEC}_N\times\_MANT\_DIG
\]

— 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,
\[
\left[ 1 + p \log_{10} 2 \right]
\]

\[
\text{FLT}_N\_DECIMAL\_DIG \\
\text{FLT}_N\times\_DECIMAL\_DIG
\]

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

\[
\text{FLT}_N\_DIG \\
\text{FLT}_N\times\_DIG
\]

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

\[
\text{FLT}_N\_MIN\_EXP \\
\text{FLT}_N\times\_MIN\_EXP \\
\text{DEC}_N\_MIN\_EXP \\
\text{DEC}_N\times\_MIN\_EXP
\]
— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\left\lfloor \log_{10} 2^{e_{\text{min}}-1} \right\rfloor$

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

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

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

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

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

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

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

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

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

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

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

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

\[
\text{FLT}_N \_\text{TRUE} \_\text{MIN} \\
\text{FLTNX}_\text{TRUE} \_\text{MIN} \\
\text{DEC}_N \_\text{TRUE} \_\text{MIN} \\
\text{DECNX}_\text{TRUE} \_\text{MIN}
\]

With the following change, \texttt{DECIMAL\_DIG} characterizes conversions of supported IEC 60559 encodings, which may be wider than supported floating types.
Change to C11 + TS18661-1 + TS18661-2:

In 5.2.4.2.2#11, change the bullet defining `DECIMAL_DIG` from:

— number of decimal digits, \( n \), such that any floating-point number in the widest supported floating type with ...

to:

— number of decimal digits, \( n \), such that any floating-point number in the widest of the supported floating types and the supported IEC 60559 encodings with ...

8 Conversions

The following change to C11 + TS18661-1 + TS18661-2 enhances the usual arithmetic conversions to handle interchange and extended floating types. IEC 60559 recommends 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. The following change supports this restriction.

Changes to C11 + TS18661-1 + TS18661-2:

Replace 6.3.1.4#1a:

[1a] When a finite value of decimal 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.

with:

[1a] 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.

Replace 6.3.1.4#2a:

[2a] When a value of integer type is converted to a decimal 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.

with:

[2a] 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.

In 6.3.1.8#1, replace the following items after “This pattern is called the usual arithmetic conversions:”:

If one operand has decimal floating type, the other operand shall not have standard floating, complex, or imaginary type.
First, if the type of either operand is _Decimal128, the other operand is converted to _Decimal128.

Otherwise, if the type of either operand is _Decimal64, the other operand is converted to _Decimal64.

Otherwise, if the type of either operand is _Decimal32, the other operand is converted to _Decimal32.

If there are no decimal floating types in the operands:

First, 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.

Otherwise, if the corresponding real type of either operand is float, the other operand is converted, without change of type domain, to a type whose corresponding real type is float.62)

with:

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.

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

Otherwise, 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 a standard floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same standard floating type.

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.
9 Constants

The following changes to C11 + TS18661-1 + TS18661-2 provide suffixes that designate constants of interchange and extended floating types.

Changes to C11 + TS18661-1 + TS18661-2:

Change floating-suffix in 6.4.4.2 from:

floating-suffix: one of
  f l F L df dd dl DF DD DL

to:

floating-suffix: one of
  f l F L df dd dl DF DD DL fN FN fNx FNx dN DN dNx DNx

Replace 6.4.4.2#2a:

[2a] A floating-suffix df, dd, dl, DF, DD, or DL shall not be used in a hexadecimal-floating-constant.

with:

[2a] A floating-suffix df, dd, dl, DF, DD, DL, dN, DN, dNx, or DNx shall not be used in a hexadecimal-floating-constant.

[2b] A floating-suffix shall not designate a type that the implementation does not provide.

Replace 6.4.4.2#4a:

[4a] If a floating constant is suffixed by df or DF, it has type _Decimal32. If suffixed by dd or DD, it has type _Decimal64. If suffixed by dl or DL, it has type _Decimal128.

with:

[4a] 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 df or DF, it has type _Decimal32. If suffixed by dd or DD, it has type _Decimal64. If suffixed by dl or DL, it has type _Decimal128. If suffixed by dN or DN, it has type _DecimalN. If suffixed by dNx or DNx, it has type _DecimalNx.

Replace the second sentence of 6.4.4.2#5a:

The quantum exponent is specified to be the same as for the corresponding strtod32, strtod64, or strtod128 function for the same numeric string.

with:

The quantum exponent is specified to be the same as for the corresponding strtodN or strtodNx function for the same numeric string.
10 Expressions

The following changes to C11 + TS18661-1 + TS18661-2 specify operator constraints for interchange and extended floating types.

Changes to C11 + TS18661-1 + TS18661-2:

Replace 6.5.5#2a:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

Replace 6.5.6#3a:

[3a] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

[3a] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

Replace 6.5.8#2a:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type.

with:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type or binary floating type.

Replace 6.5.9#2a:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

Replace 6.5.15#3a:

[3a] If either of the second or third operands has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.
with:

[3a] 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.

Replace 6.5.16#2a:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, complex type, or imaginary type.

with:

[2a] If either operand has decimal floating type, the other operand shall not have standard floating type, binary floating type, complex type, or imaginary type.

In F.9.2#1, replace the first sentence:

[1] The equivalences noted below apply to expressions of standard floating types.

with:

[1] The equivalences noted below apply to expressions of standard floating types and binary floating types.

11 Non-arithmetic interchange formats

An implementation supports IEC 60559 arithmetic interchange formats by providing the corresponding interchange floating types. An implementation supports IEC 60559 non-arithmetic formats by providing the encoding-to-encoding conversion functions in `<math.h>` and the string-to-encoding and string-from-encoding functions in `<stdlib.h>`. See 6.2.5. These functions, together with functions required for interchange floating types, provide conversions between any two of the supported IEC 60559 arithmetic and non-arithmetic interchange formats and between character sequences and any supported IEC 60559 arithmetic or non-arithmetic format.

12 Mathematics `<math.h>`

This clause specifies changes to C11 + TS18661-1 + TS18661-2 to include functions and macros for interchange and extended floating types. The binary types are supported by functions and macros corresponding to those specified for standard floating types (`float`, `double`, and `long double`) in C11 + TS18661-1, including Annex F. The decimal types are supported by functions and macros corresponding to those specified for decimal floating types in TS18661-2.

All classification (7.12.3) and comparison (7.12.14) macros specified in C11 + TS18661-1 + TS18661-2 naturally extend to handle interchange and extended floating types.

This clause also specifies encoding conversion functions that are part of support for the non-arithmetic interchange formats in IEC 60559 (see 6.2.5).

Changes to C11 + TS18661-1 + TS18661-2:

In 7.12#1, change the second sentence from:

Most synopses specify a family of functions consisting of a principal function with one or more `double` parameters, a `double` return value, or both; and other functions with the same name
but with f and l suffixes, which are corresponding functions with float and long double parameters, return values, or both.

Most synopses specify a family of functions consisting of:

- a principal function with one or more double parameters, a double return value, or both; and,
- other functions with the same name but with f, l, fN, fNx, dN, and dNx suffixes, which are corresponding functions whose parameters, return values, or both are of types float, long double, _FloatN, _FloatNx, _DecimalN, and _DecimalNx, respectively.

After 7.12#1d, add:

[1e] 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.

Change 7.12#2, from:

[2] The types

    float_t
double_t

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

to:

[2] The types

    float_t
double_t
long_double_t

and for each supported type _FloatN, the type

    _FloatN_t

and for each supported type _DecimalN, the type

    _DecimalN_t
are floating types, such that:

— each of the types has at least the range and precision of the corresponding real floating type `float`, `double`, `long double`, `_FloatN`, and `_DecimalN`, respectively;

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

— `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` 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 5.2.4.2.2a to be used for evaluating operations and constants of that standard or binary floating type. If `DEC_EVAL_METHOD` is nonnegative, then each of the types corresponding to a decimal floating type is the type whose range and precision are specified by 5.2.4.2.2a to be used for evaluating operations and constants of that decimal floating type.

Delete footnote 227:

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

12.1 Macros

Changes to C11 + TS18661-1 + TS18661-2:

Replace 7.12#3a:

[3a] The macro

```
HUGE_VAL_D32
```

expands to a constant expression of type `_Decimal64` representing positive infinity. The macros

```
HUGE_VAL_D64
HUGE_VAL_D128
```

are respectively `_Decimal64` and `_Decimal128` analogues of `HUGE_VAL_D32`. 
with:

[3a] The macros

\[
\begin{align*}
\text{HUGE\_VAL\_FN} \\
\text{HUGE\_VAL\_DN} \\
\text{HUGE\_VAL\_FNX} \\
\text{HUGE\_VAL\_DNX}
\end{align*}
\]

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

Replace 7.12#5b:

[5b] The decimal signaling NaN macros

\[
\begin{align*}
\text{SNAND32} \\
\text{SNAND64} \\
\text{SNAND128}
\end{align*}
\]

each expands to a constant expression of the respective decimal floating type representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.

with:

[5b] The signaling NaN macros

\[
\begin{align*}
\text{SNANFN} \\
\text{SNANDN} \\
\text{SNANFX} \\
\text{SNANDNX}
\end{align*}
\]

expand to constant expressions of types \texttt{\_FloatN}, \texttt{\_DecimalN}, \texttt{\_FloatNx}, and \texttt{\_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.

Replace 7.12#7b:

[7b] The macros

\[
\begin{align*}
\text{FP\_FAST\_FMAD32} \\
\text{FP\_FAST\_FMAD64} \\
\text{FP\_FAST\_FMAD128}
\end{align*}
\]

are, respectively, \texttt{\_Decimal32}, \texttt{\_Decimal64}, and \texttt{\_Decimal128} analogues of \texttt{FP\_FAST\_FMA}.
with:

[7b] The macros

\[
\begin{align*}
&\text{FP\_FAST\_FMAF}\_N \\
&\text{FP\_FAST\_FMAD}\_N \\
&\text{FP\_FAST\_FMAF}\_N \times \\
&\text{FP\_FAST\_FMAD}\_N \\
\end{align*}
\]

are, respectively, \_Float\_N, \_Decimal\_N, \_Float\_Nx, and \_Decimal\_Nx analogues of \text{FP\_FAST\_FMA}.

Replace 7.12#7c:

[7c] The macros

\[
\begin{align*}
&\text{FP\_FAST\_D32ADDD64} \\
&\text{FP\_FAST\_D32ADDD128} \\
&\text{FP\_FAST\_D64ADDD128} \\
&\text{FP\_FAST\_D32SUBD64} \\
&\text{FP\_FAST\_D32SUBD128} \\
&\text{FP\_FAST\_D64SUBD128} \\
&\text{FP\_FAST\_D32MULD64} \\
&\text{FP\_FAST\_D32MULD128} \\
&\text{FP\_FAST\_D64MULD128} \\
&\text{FP\_FAST\_D32DIVD64} \\
&\text{FP\_FAST\_D32DIVD128} \\
&\text{FP\_FAST\_D64DIVD128} \\
&\text{FP\_FAST\_D32FMAD64} \\
&\text{FP\_FAST\_D32FMAD128} \\
&\text{FP\_FAST\_D64FMAD128} \\
&\text{FP\_FAST\_D32SQRTD64} \\
&\text{FP\_FAST\_D32SQRTD128} \\
&\text{FP\_FAST\_D64SQRTD128} \\
\end{align*}
\]

are decimal analogues of \text{FP\_FAST\_FADD}, \text{FP\_FAST\_FADDL}, \text{FP\_FAST\_DADDL}, etc.

with:

[7c] 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.
[7d] For $M < N$, the macros

```
FP_FAST_FMADDN
FP_FAST_FMSUBN
FP_FAST_FMULN
FP_FAST_FDIVN
FP_FAST_FMAFN
FP_FAST_FMSQRTFN
FP_FAST_DADDN
FP_FAST_DSUBD
FP_FAST_DMDIVD
FP_FAST_DFMADDN
FP_FAST_DMSQRTDN
```

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 $M$.

[7e] For $M \leq N$, the macros

```
FP_FAST_FMADDN
FP_FAST_FMSUBN
FP_FAST_FMULN
FP_FAST_FDIVN
FP_FAST_FMAFN
FP_FAST_FMSQRTFN
FP_FAST_DADDN
FP_FAST_DSUBD
FP_FAST_DMDIVD
FP_FAST_DFMADDN
FP_FAST_DMSQRTDN
```

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 $M$. 
[7f] For \( M < N \), the macros

\[
\begin{align*}
\text{FP\_FAST\_FMXADD\_FN} \\
\text{FP\_FAST\_FMXSUB\_FN} \\
\text{FP\_FAST\_FMXMUL\_FN} \\
\text{FP\_FAST\_FMXDIV\_FN} \\
\text{FP\_FAST\_FMXFMA\_FN} \\
\text{FP\_FAST\_FMXSQRT\_FN} \\
\text{FP\_FAST\_DMXADD\_DN} \\
\text{FP\_FAST\_DMXSUB\_DN} \\
\text{FP\_FAST\_DMXMUL\_DN} \\
\text{FP\_FAST\_DMXDIV\_DN} \\
\text{FP\_FAST\_DMXFMA\_DN} \\
\text{FP\_FAST\_DMXSQRT\_DN}
\end{align*}
\]

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 \( M \).

[7g] For \( M < N \), the macros

\[
\begin{align*}
\text{FP\_FAST\_FMXADD\_FNX} \\
\text{FP\_FAST\_FMXSUB\_FNX} \\
\text{FP\_FAST\_FMXMUL\_FNX} \\
\text{FP\_FAST\_FMXDIV\_FNX} \\
\text{FP\_FAST\_FMXFMA\_FNX} \\
\text{FP\_FAST\_FMXSQRT\_FNX} \\
\text{FP\_FAST\_DMXADD\_DNX} \\
\text{FP\_FAST\_DMXSUB\_DNX} \\
\text{FP\_FAST\_DMXMUL\_DNX} \\
\text{FP\_FAST\_DMXDIV\_DNX} \\
\text{FP\_FAST\_DMXFMA\_DNX} \\
\text{FP\_FAST\_DMXSQRT\_DNX}
\end{align*}
\]

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 \( M \).

### 12.2 Floating-point environment

**Changes to C11 + TS18661-1 + TS18661-2:**

In 7.6.1a#2, change the first sentence from:

> The **FENV\_ROUND** pragma provides a means to specify a constant rounding direction for floating-point operations for standard floating types within a translation unit or compound statement.

To:

> The **FENV\_ROUND** pragma provides a means to specify a constant rounding direction for floating-point operations for standard floating types within a translation unit or compound statement.
The **FENV_ROUND** pragma provides a means to specify a constant rounding direction for floatiing-point operations for standard and binary floating types within a translation unit or compound statement.

In 7.6.1a#3, change the first sentence from:

> direction shall be one of the names of the supported rounding direction macros for operations for standard floating types (7.6), or **FE_DYNAMIC**.

To:

> direction shall be one of the names of the supported rounding direction macros for use with **fegetround** and **fesetround** (7.6), or **FE_DYNAMIC**.

In 7.6.1a#4, change the first sentence from:

> The **FENV_ROUND** directive affects operations for standard floating types.

To:

> The **FENV_ROUND** directive affects operations for standard and binary floating types.

In 7.6.1a#4, change the table title from:

> Functions affected by constant rounding modes – for standard floating types

To:

> Functions affected by constant rounding modes – for standard and binary floating types

In 7.6.1a#4, replace the sentence following the table:

> Each `<math.h>` function listed in the table above indicates the family of functions of all standard floating types (for example, **acosf** and **acosl** as well as **acos**).

With:

> Each `<math.h>` function listed in the table above indicates the family of functions of all standard and binary floating types (for example, **acosf**, **acosl**, **acosfN**, and **acosfNx** as well as **acos**).

After 7.6.1a#4, add:

> [4a] The **fMencfN**, **strfromencfN**, and **strtoencfN** functions for binary interchange types are also affected by constant rounding modes.

In 7.6.1b#2 after the table, add:

> Each `<math.h>` function listed in the table above indicates the family of functions of all decimal floating types (for example, **acosdNx**, as well as **acosdN**).
After 7.6.1b#2, add:

[3] The $d\text{Mencbind}N$, $d\text{Mencdecd}N$, $\text{strfromencbind}N$, $\text{strfromencdecd}N$, $\text{strtoencbind}N$, and $\text{strtoencdecd}N$ functions for decimal interchange types are also affected by constant rounding modes.

Change 7.6.3 from:

The $\text{fegetround}$ and $\text{fesetround}$ functions provide control of rounding direction modes.

to:

The functions in this subclause provide control of rounding direction modes.

Change 7.6.3.1#2 from:

The $\text{fegetround}$ function gets the current value of the dynamic rounding direction mode.

to:

The $\text{fegetround}$ function gets the current value of the dynamic rounding direction mode for operations for standard and binary floating types.

In 7.6.3.2#2, change the first sentence from:

The $\text{fesetround}$ function sets the dynamic rounding direction mode to the rounding direction represented by its argument $\text{round}$.

to:

The $\text{fesetround}$ function sets the dynamic rounding direction mode to the rounding direction represented by its argument $\text{round}$ for operations for standard and binary floating types.

12.3 Functions

Changes to C11 + TS18661-1 + TS18661-2:

Add the following list of function prototypes to the synopsis of the respective subclauses. In each synopsis where a prototype with a $dN$ suffix is added, remove any prototypes with a $d32$, $d64$, or $d128$ suffix.

7.12.4 Trigonometric functions

```c
_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 acoshfN(_FloatN x);
_FloatN acoshfN(_FloatN x);
_FloatN acoshfN(_FloatN x);
_FloatN acoshdN(_DecimalN x);
_FloatN acoshdN(_DecimalN x);
_FloatN asinhfN(_FloatN x);
_FloatN asinhfN(_FloatN x);
_FloatN asinhfN(_FloatN x);
_FloatN asinhdN(_DecimalN x);
_FloatN asinhdN(_DecimalN x);
_FloatN atanhfN(_FloatN x);
_FloatN atanhfN(_FloatN x);
_FloatN atanhfN(_FloatN x);
_FloatN atanhfN(_FloatN x);
_FloatN atanhdN(_DecimalN x);
_FloatN atanhdN(_DecimalN x);
_FloatN atanhdN(_DecimalN x);
_FloatN atanhdN(_DecimalN x);
_FloatN coshfN(_FloatN x);
_FloatN coshfN(_FloatN x);
_FloatN coshfN(_FloatN x);
_FloatN coshfN(_FloatN x);
7.12.6 Exponential and logarithmic functions

_FloatN sinhF(_FloatN x);
_FloatN sinhF(_FloatN x);
_DecimalN sinhD(_DecimalN x);
_DecimalN sinhD(_DecimalN x);
_FloatN tanhF(_FloatN x);
_FloatN tanhF(_FloatN x);
_DecimalN tanhD(_DecimalN x);
_DecimalN tanhD(_DecimalN x);

_FloatN expF(_FloatN x);
_FloatN expF(_FloatN x);
_DecimalN expD(_DecimalN x);
_DecimalN expD(_DecimalN x);
_FloatN exp2F(_FloatN x);
_FloatN exp2F(_FloatN x);
_DecimalN exp2D(_DecimalN x);
_DecimalN exp2D(_DecimalN x);
_FloatN expm1F(_FloatN x);
_FloatN expm1F(_FloatN x);
_DecimalN expm1D(_DecimalN x);
_DecimalN expm1D(_DecimalN x);
_FloatN frexF(_FloatN value, int *exp);
_FloatN frexF(_FloatN value, int *exp);
_DecimalN frexD(_DecimalN value, int *exp);
_DecimalN frexD(_DecimalN value, int *exp);
_int ilogBF(_FloatN x);
_int ilogBF(_FloatN x);
_int ilogBD(_DecimalN x);
_int ilogBD(_DecimalN x);
_FloatN ldexpF(_FloatN value, int exp);
_FloatN ldexpF(_FloatN value, int exp);
_DecimalN ldexpD(_DecimalN value, int exp);
_DecimalN ldexpD(_DecimalN value, int exp);
_long int llogBF(_FloatN x);
_long int llogBF(_FloatN x);
_long int llogBD(_DecimalN x);
_long int llogBD(_DecimalN x);
7.12.7 Power and absolute-value functions

_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 log1pfN(_FloatN x);
_FloatNx log1pfNx(_FloatNx x);
_DecimalN log1pdN(_DecimalN x);
_DecimalNx log1pdNx(_DecimalNx x);

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

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

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

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

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

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

_FloatN fabsfN(_FloatN x);
_FloatNx fabsfNx(_FloatNx x);
_DecimalN fabsdN(_DecimalN x);
_DecimalNx fabsdNx(_DecimalNx x);
7.12.8 Error and gamma functions

_FloatN erffN(_FloatN x);  
_FloatNx erffNx(_FloatNx x);  
DecimalN erfDN(_DecimalN x);  
DecimalNx erfDNx(_DecimalNx x);

_FloatN erfcfN(_FloatN x);  
_FloatNx erfcfNx(_FloatNx x);  
DecimalN erfcDN(_DecimalN x);  
DecimalNx erfcDNx(_DecimalNx x);

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

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

7.12.9 Nearest integer functions

_FloatN ceilfN(_FloatN x);  
_FloatNx ceilfNx(_FloatNx x);  
DecimalN ceilDN(_DecimalN x);  
DecimalNx ceilDNx(_DecimalNx x);

_FloatN floorfN(_FloatN x);  
_FloatNx floorfNx(_FloatNx x);  
DecimalN floorDN(_DecimalN x);  
DecimalNx floorDNx(_DecimalNx x);
_FloatN nearbyintfN(_FloatN x);
_FloatNx nearbyintfNx(_FloatNx x);
_DecimalN nearbyintdN(_DecimalN x);
_DecimalNx nearbyintdNx(_DecimalNx x);

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

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

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

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

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

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

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

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

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

_FloatN fmodf(_FloatN x, _FloatN y);
_FloatNx fmodfNx(_FloatNx x, _FloatNx y);
_DecimalN fmodd(_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 remquof(_FloatN x, _FloatN y, int *quo);
_FloatNx remquofNx(_FloatNx x, _FloatNx y, int *quo);

7.12.11 Manipulation functions

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

_FloatN nanf(const char *tagp);
_FloatNx nanfNx(const char *tagp);
_DecimalN nand(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);
7.12.12 Maximum, minimum, and positive difference functions

_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);

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

void encodedecdN(unsigned char * restrict encptr,
               const _DecimalN * restrict xptr);
void decodedecdN(_DecimalN * restrict xptr,
                 const unsigned char * restrict encptr);
void encodebindN(unsigned char * restrict encptr,
               const _DecimalN * restrict xptr);
void decodebindN(_DecimalN * restrict xptr,
                 const unsigned char * restrict encptr);
7.12.13 Floating multiply-add

\[
\begin{align*}
\text{float}N & \text{fminf}(\text{float}N x, \text{float}N y); \quad / \ M < \ N \\
\text{float}N & \text{fminfn}(\text{float}N x, \text{float}N y); \quad / \ M < \ \leq \ N \\
\text{decimal}N & \text{fmin}(\text{decimal}N x, \text{decimal}N y); \\
\text{decimal}N & \text{fminfn}(\text{decimal}N x, \text{decimal}N y); \\
\text{float}N & \text{fmaxmagf}(\text{float}N x, \text{float}N y); \\
\text{float}N & \text{fmaxmagfn}(\text{float}N x, \text{float}N y); \\
\text{decimal}N & \text{fmaxmag}(\text{decimal}N x, \text{decimal}N y); \\
\text{decimal}N & \text{fmaxmagdn}(\text{decimal}N x, \text{decimal}N y); \\
\text{float}N & \text{fminmagf}(\text{float}N x, \text{float}N y); \\
\text{float}N & \text{fminmagfn}(\text{float}N x, \text{float}N y); \\
\text{decimal}N & \text{fminmag}(\text{decimal}N x, \text{decimal}N y); \\
\text{decimal}N & \text{fminmagdn}(\text{decimal}N x, \text{decimal}N y); \\
\end{align*}
\]

7.12.14 Functions that round result to narrower format

\[
\begin{align*}
\text{float}M & \text{fmaff}(\text{float}N x, \text{float}N y, \text{float}N z); \quad / \ M < \ N \\
\text{float}M & \text{fmaffn}(\text{float}N x, \text{float}N y, \text{float}N z); \quad / \ \leq \ M \leq \ N \\
\text{decimal}N & \text{fmad}(\text{decimal}N x, \text{decimal}N y, \text{decimal}N z); \\
\text{decimal}N & \text{fmadn}(\text{decimal}N x, \text{decimal}N y, \text{decimal}N z); \\
\text{float}M & \text{fmaddf}(\text{float}N x, \text{float}N y); \quad / \ M < \ \leq \ N \\
\text{float}M & \text{fmaddfn}(\text{float}N x, \text{float}N y); \quad / \ \leq \ M \leq \ \leq \ N \\
\text{decimal}M & \text{dmad}(\text{decimal}N x, \text{decimal}N y); \quad / \ M < \ \leq \ N \\
\text{decimal}M & \text{dmadn}(\text{decimal}N x, \text{decimal}N y); \quad / \ \leq \ M \leq \ \leq \ N \\
\text{float}M & \text{fmsubf}(\text{float}N x, \text{float}N y); \quad / \ M < \ N \\
\text{float}M & \text{fmsubfn}(\text{float}N x, \text{float}N y); \quad / \ \leq \ M \leq \ \leq \ N \\
\text{decimal}M & \text{dmsub}(\text{decimal}N x, \text{decimal}N y); \quad / \ M < \ \leq \ N \\
\text{decimal}M & \text{dmsubn}(\text{decimal}N x, \text{decimal}N y); \quad / \ \leq \ M \leq \ \leq \ N \\
\text{float}M & \text{fmmulf}(\text{float}N x, \text{float}N y); \quad / \ M < \ N \\
\text{float}M & \text{fmmulfn}(\text{float}N x, \text{float}N y); \quad / \ \leq \ M \leq \ \leq \ N \\
\text{decimal}M & \text{dmuldn}(\text{decimal}N x, \text{decimal}N y); \quad / \ M < \ \leq \ N \\
\text{decimal}M & \text{dmuldn}(\text{decimal}N x, \text{decimal}N y); \quad / \ \leq \ M \leq \ \leq \ N \\
\text{decimal}M & \text{dmmuldn}(\text{decimal}N x, \text{decimal}N y); \quad / \ M < \ \leq \ N \\
\end{align*}
\]
F.10.12 Total order functions

```c
int totalorderfN(_FloatN x, _FloatN y);
int totalorderfNx(_FloatN x, _FloatN y);
int totalorderdN(_DecimalN x, _DecimalN y);
int totalorderdNx(_DecimalN x, _DecimalN y);

int totalordermagfN(_FloatN x, _FloatN y);
int totalordermagfNx(_FloatN x, _FloatN y);
int totalordermagdN(_DecimalN x, _DecimalN y);
int totalordermagdNx(_DecimalN x, _DecimalN y);
```
F.10.13 Payload functions

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

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

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

In 7.12.6.4#2, change the third sentence from:

> If the type of the function is a standard floating type, the exponent is an integral power of 2.

to:

> If the type of the function is a standard or binary floating type, the exponent is an integral power of 2.

In 7.12.6.4#3, change the second sentence from:

> Otherwise, the `frexp` functions return the value `x`, such that: `x` has a magnitude in the interval \([1/2, 1)\) or zero, and `value` equals `x \times 2^{\exp}`, when the type of the function is a standard floating type; ...


to:

> Otherwise, the `frexp` functions return the value `x`, such that: `x` has a magnitude in the interval \([1/2, 1)\) or zero, and `value` equals `x \times 2^{\exp}`, when the type of the function is a standard or binary floating type; ...

In 7.12.6.6#2, change the first sentence from:

> The `ldexp` functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard floating type, or by an integral power of 10 when the type of the function is a decimal floating type.


to:

> The `ldexp` functions multiply a floating-point number by an integral power of 2 when the type of the function is a standard or binary floating type, or by an integral power of 10 when the type of the function is a decimal floating type.
Change 7.12.6.6#3 from:

[3] The ldexp functions return $x \times 2^{\text{exp}}$ when the type of the function is a standard floating type, or return $x \times 10^{\text{exp}}$ when the type of the function is a decimal floating type.

to:

[3] The ldexp functions return $x \times 2^{\text{exp}}$ when the type of the function is a standard or binary floating type, or return $x \times 10^{\text{exp}}$ when the type of the function is a decimal floating type.

In 7.12.6.11#2, change the second sentence from:

If $x$ is subnormal it is treated as though it were normalized; thus, for positive finite $x$,  
\[ 1 \leq x \times b^{-\log_b(x)} < b \]

where $b = \text{FLT\_RADIX}$ if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type.

to:

If $x$ is subnormal it is treated as though it were normalized; thus, for positive finite $x$,  
\[ 1 \leq x \times b^{-\log_b(x)} < b \]

where $b = \text{FLT\_RADIX}$ if the type of the function is a standard floating type, $b = 2$ if the type of the function is a binary floating type, or $b = 10$ if the type of the function is a decimal floating type.

In 7.12.6.13#2, change the first sentence from:

The scalbn and scalbln functions compute $x \times b^n$, where $b = \text{FLT\_RADIX}$ if the type of the function is a standard floating type, or $b = 10$ if the type of the function is a decimal floating type.

to:

The scalbn and scalbln functions compute $x \times b^n$, where $b = \text{FLT\_RADIX}$ if the type of the function is a standard floating type, $b = 2$ if the type of the function is a binary floating type, or $b = 10$ if the type of the function is a decimal floating type.

### 12.4 Encoding conversion functions

The functions in this subclause, together with the numerical conversion functions for encodings in clause 13, support the non-arithmetic interchange formats specified by IEC 60559.
Changes to C11 + TS18661-1 + TS18661-2:

After 7.12.11.7, add:

7.12.11.7a 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


7.12.11.7b 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


7.12.11.7c Encoding-to-encoding conversion functions

[1] An implementation shall declare a 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 dMdecfN and dMencbindN functions for each M and N equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.
7.12.11.7c.1 The \texttt{fMencfN} functions

Synopsis

[1] \texttt{#include <math.h>}

\begin{verbatim}
void fMencfN(unsigned char * restrict encMptr,
             const unsigned char * restrict encNptr);
\end{verbatim}

Description

[2] These functions convert between IEC 60559 binary interchange formats. These functions interpret the \(N/8\) element array pointed to by \texttt{encNptr} as an encoding of width \(N\) bits. They convert the encoding to an encoding of width \(M\) bits and store the resulting encoding as an \(M/8\) element array in the object pointed to by \texttt{encMptr}. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

7.12.11.7c.2 The \texttt{dMencdecN} and \texttt{dMencbindN} functions

Synopsis

[1] \texttt{#include <math.h>}

\begin{verbatim}
void dMencdecN(unsigned char * restrict encMptr,
               const unsigned char * restrict encNptr);
void dMencbindN(unsigned char * restrict encMptr,
                const unsigned char * restrict encNptr);
\end{verbatim}

Description

[2] These functions convert between IEC 60559 decimal interchange formats that use the same encoding scheme. The \texttt{dMencdecN} functions convert between formats using the encoding scheme based on decimal encoding of the significand. The \texttt{dMencbindN} functions convert between formats using the encoding scheme based on binary encoding of the significand. These functions interpret the \(N/8\) element array pointed to by \texttt{encNptr} as an encoding of width \(N\) bits. They convert the encoding to an encoding of width \(M\) bits and store the resulting encoding as an \(M/8\) element array in the object pointed to by \texttt{encMptr}. The conversion rounds and raises floating-point exceptions as specified in IEC 60559. The order of bytes in the arrays is implementation-defined.

Returns

[3] These functions return no value.

In F.3, change the row:

| convertFormat – different formats | cast and implicit conversions | 6.3.1.5, 6.5.4 |
After F.3 [3], add:

[3a] C operations provide the `convertFormat` operations for the different kinds of IEC 60559 formats as follows:

<table>
<thead>
<tr>
<th>convertFormat - different formats</th>
<th>cast, implicit conversions, conversion functions (details below)</th>
<th>6.3.1.5, 6.5.4, 7.12.11, 7.22.11b, 7.22.1 (details below)</th>
</tr>
</thead>
</table>

13 Numeric conversion functions in `<stdlib.h>`

This clause specifies functions to convert between character sequences and the interchange and extended floating types. Conversions from character sequences are provided by functions analogous to the `strtod` function in `<stdlib.h>`. Conversions to character sequences are provided by functions analogous to the `strfromd` function in `<stdlib.h>`.

This clause also specifies functions to convert between character sequences and IEC 60559 interchange format encodings.

Changes to C11 + TS18661-1 + TS18661-2:

After 7.22.1#1, insert

[1a] For each interchange or extended floating type that the implementation provides, `<stdlib.h>` shall declare the associated functions. Conversely, for each such type that the
implementation does not provide, `<stdlib.h>` shall not declare the associated functions
unless specified otherwise.

Replace 7.22.1.2a and 7.22.1.2b:

7.22.1.2a The `strfromd`, `strfromf`, and `strfroml` functions

Synopsis

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

Description

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

Returns

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

7.22.1.2b The `strfromdN` functions

Synopsis

[1] #define __STDC_WANT_IEC_60559_DFP_EXT__
    #include <stdlib.h>
    int strfromd32(char * restrict s, size_t n,
                   const char * restrict format, _Decimal32 fp);
    int strfromd64(char * restrict s, size_t n,
                   const char * restrict format, _Decimal64 fp);
    int strfromd128(char * restrict s, size_t n,
                    const char * restrict format, _Decimal128 fp);

Description

[2] The `strfromdN` functions are equivalent to `snprintf(s, n, format, fp)` (7.21.6.5), except the `format` string contains only the character `%`, an optional precision that does not contain an asterisk `*`, and one of the conversion specifiers `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, which applies to the type (`_Decimal32`, `_Decimal64`, or `_Decimal128`) indicated by the
function suffix (rather than by a length modifier). Use of these functions with any other
\texttt{format} string results in undefined behavior.

\textbf{Returns}

\textbf{[3]} The \texttt{strfromdN} functions return the number of characters that would have been written
had \texttt{n} been sufficiently large, not counting the terminating null character. Thus, the null-
terminated output has been completely written if and only if the returned value is less than \texttt{n}.

with:

\textbf{7.22.1.2a The \texttt{strfromd, strfromf, strfroml, strfromfN, strfromfNx, strfromdN, and strfromdNx} functions}

\textbf{Synopsis}

\textbf{[1]} \#include <stdlib.h>

\begin{verbatim}
int strfromd (char * restrict s, size_t n,
            const char * restrict format, double fp);
int strfromf (char * restrict s, size_t n,
            const char * restrict format, float fp);
int strfroml (char * restrict s, size_t n,
            const char * restrict format, long double fp);
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);
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);
\end{verbatim}

\textbf{Description}

\textbf{[2]} The \texttt{strfromd, strfromf, strfroml, strfromfN, strfromfNx, strfromdN, and
\texttt{strfromdNx}} functions are equivalent to \texttt{snprintf(s, n, format, fp)} \texttt{(7.21.6.5)},
extcept the \texttt{format} string contains only the character \%, an optional precision that does not
contain an asterisk *, and one of the conversion specifiers \texttt{a, A, e, E, f, F, g, or G}, which applies
to the type \texttt{(double, float, long double, _FloatN, _FloatNx, _DecimalN, or
_DecimalNx)} indicated by the function suffix (rather than by a length modifier). Use of these
functions with any other \texttt{format} string results in undefined behavior.

\textbf{Returns}

\textbf{[3]} These functions return the number of characters that would have been written had \texttt{n} been
sufficiently large, not counting the terminating null character. Thus, the null-terminated output
has been completely written if and only if the returned value is less than \texttt{n}. 
Change the first part of 7.22.1.3:

7.22.1.3 The `strtod`, `strtof`, and `strtold` functions

Synopsis

[1] `#include <stdlib.h>`

```c
double strtod(const char * restrict nptr,
              char ** restrict endptr);
float strtof(const char * restrict nptr,
             char ** restrict endptr);
long double strtold (const char * restrict nptr,
                    char ** restrict endptr);

strtofN(strtofN, N)
strtofNx(strtofNx, N)
```

Description

[2] The `strtod`, `strtof`, and `strtold` functions convert the initial portion of the string pointed to by `nptr` to `double`, `float`, and `long double` representation, respectively.

to:

7.22.1.3 The `strtod`, `strtof`, `strtold`, `strtofN`, and `strtofNx` functions

Synopsis

[1] `#include <stdlib.h>`

```c
double strtod(const char * restrict nptr,
              char ** restrict endptr);
float strtof(const char * restrict nptr,
             char ** restrict endptr);
long double strtold (const char * restrict nptr,
                    char ** restrict endptr);
_FloatN(strtofN(const char * restrict nptr,
                char ** restrict endptr);
_FloatNx(strtofNx(const char * restrict nptr,
                 char ** restrict endptr);
```

Description

[2] The `strtod`, `strtof`, `strtold`, `strtofN`, and `strtofNx` functions convert the initial portion of the string pointed to by `nptr` to `double`, `float`, `long double`, `_FloatN`, and `_FloatNx` representation, respectively.

Change 7.22.1.3 #5:

[5] If the subject sequence has the hexadecimal form and `FLT_RADIX` is a power of 2, the value resulting from the conversion is correctly rounded.

to:

[5] If the subject sequence has the hexadecimal form and the radix of the return type of the function is a power of 2, the value resulting from the conversion is correctly rounded.
In 7.22.1.3 #8, change the first sentence:

[8] If the subject sequence has the hexadecimal form, \texttt{FLT\_RADIX} is not a power of 2, ...

to:

[8] If the subject sequence has the hexadecimal form, the radix of the return type of the function is not a power of 2, and ...

In 7.22.1.3 #10, in the third sentence, change:

plus or minus \texttt{HUGE\_VAL}, \texttt{HUGE\_VALF}, or \texttt{HUGE\_VALL} is returned

to:

plus or minus \texttt{HUGE\_VAL}, \texttt{HUGE\_VALF}, \texttt{HUGE\_VALL}, \texttt{HUGE\_VALFN}, or \texttt{HUGE\_VALFNX} is returned

In 7.22.1.3a, change:

\textbf{Synopsis}

[1] \#define \_STDC\_WANT\_IEC\_60559\_DFP\_EXT\_
#include <stdlib.h>

\_Decimal32 \texttt{strtod32} (const char \* restrict nptr,
char ** restrict endptr);
\_Decimal64 \texttt{strtod64} (const char \* restrict nptr,
char ** restrict endptr);
\_Decimal128 \texttt{strtod128} (const char \* restrict nptr,
char ** restrict endptr);

to:

\textbf{Synopsis}

[1] \#include <stdlib.h>

\_Decimal\textit{N} \texttt{strtod\textit{N}} (const char \* restrict nptr,
char ** restrict endptr);
\_Decimal\textit{Nx} \texttt{strtod\textit{Nx}} (const char \* restrict nptr,
char ** restrict endptr);

After 7.22.1.3a, insert:

\textbf{7.22.1.3b String-to-encoding functions}

[1] An implementation shall declare the \texttt{strtoenc\textit{N}} function for each \textit{N} equal to the width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An implementation shall declare both the \texttt{strtoencoded\textit{N}} and \texttt{strtoencbind\textit{N}} functions for each \textit{N} equal to the width of a supported IEC 60559 arithmetic or non-arithmetic decimal interchange format.
7.22.1.3b.1 The strtoencf$N$ functions

Synopsis

[1] #include <stdlib.h>

    void strtoencf$N$(unsigned char * restrict encptr,
                    const char * restrict nptr, char ** restrict endptr);

Description

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

Returns

[3] These functions return no value.

7.22.1.3b.2 The strtoencdecd$N$ and strtoencbind$N$ functions

Synopsis

[1] #include <stdlib.h>

    void strtoencdecd$N$(unsigned char * restrict encptr,
                         const char * restrict nptr, char ** restrict endptr);
    void strtoencbind$N$(unsigned char * restrict encptr,
                         const char * restrict nptr, char ** restrict endptr);

Description

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

Returns

[3] These functions return no value.

7.22.1.3c String-from-encoding functions

[1] An implementation shall declare the strfromencf$N$ function for each $N$ equal to the
width of a supported IEC 60559 arithmetic or non-arithmetic binary interchange format. An
implementation shall declare both the strfromencdecd$N$ and strfromencbind$N$
functions for each $N$ equal to the width of a supported IEC 60559 arithmetic or non-arithmetic
decimal interchange format.
7.22.1.3c.1 The `strfromencfN` 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 `strfromencfN` functions are similar to the `strfromfN` functions, except the input is the value of the \( N/8 \) element array pointed to by `encptr`, interpreted as an IEC 60559 binary\( N \) encoding. The order of bytes in the arrays is implementation-defined.

Returns

[3] The `strfromencfN` functions return the same values as corresponding `strfromfN` functions.

7.22.1.3c.2 The `strfromencdecdN` and `strfromencbindN` functions

Synopsis

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

Description

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

Returns

[3] The `strfromencdecdN` and `strfromencbindN` functions return the same values as corresponding `strfromdN` functions.

14 Complex arithmetic `<complex.h>`

This clause specifies complex functions for corresponding real types that are interchange and extended floating types.
Changes to C11 + TS18661-1 + TS18661-2:

Change 7.3.1#3 from:

[3] Each synopsis specifies a family of functions consisting of a principal function with one or more double complex parameters and a double complex or double return value; and other functions with the same name but with f and l suffixes which are corresponding functions with float and long double parameters and return values.

to:

[3] Each synopsis specifies a family of functions consisting of:

a principal function with one or more double complex parameters and a double complex or double return value; and,

other functions with the same name but with f, l, fN, and fNx suffixes which are corresponding functions whose parameters and return values have corresponding real types float, long double, _FloatN, and _FloatNx.

Add after 7.3.1#3:

[3a] For each interchange or extended 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.

Add the following list of function prototypes to the synopsis of the respective subclauses:

7.3.5 Trigonometric functions

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

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

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

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

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

_FloatN complex ctanfN(_FloatN complex z);
_FloatNx complex ctanfNx(_FloatNx complex z);
7.3.6 Hyperbolic functions

```c
_FloatN complex cacoshf(_FloatN complex z);
_FloatN complex cacoshfN(_FloatNx complex z);
_FloatN complex casinhf(_FloatN complex z);
_FloatN complex casinhfN(_FloatNx complex z);
_FloatN complex catanhf(_FloatN complex z);
_FloatN complex catanhfN(_FloatNx complex z);
_FloatN complex ccoshf(_FloatN complex z);
_FloatN complex ccoshfN(_FloatNx complex z);
_FloatN complex csinhf(_FloatN complex z);
_FloatN complex csinhfN(_FloatNx complex z);
_FloatN complex ctanhf(_FloatN complex z);
_FloatN complex ctanhfN(_FloatNx complex z);
```

7.3.7 Exponential and logarithmic functions

```c
_FloatN complex cexpf(_FloatN complex z);
_FloatN complex cexpfN(_FloatNx complex z);
_FloatN complex clogf(_FloatN complex z);
_FloatN complex clogfN(_FloatNx complex z);
```

7.3.8 Power and absolute value functions

```c
_FloatN cabsf(_FloatN complex z);
_FloatN cabsfN(_FloatNx complex z);
_FloatN complex cpowf(_FloatN complex x, _FloatN complex y);
_FloatN complex cpowfN(_FloatNx complex x, _FloatNx complex y);
_FloatN complex csqrtf(_FloatN complex z);
_FloatN complex csqrtfN(_FloatNx complex z);
```

7.3.9 Manipulation functions

```c
_FloatN cargf(_FloatN complex z);
_FloatN cargfN(_FloatNx complex z);
_FloatN cimagf(_FloatN complex z);
_FloatN cimagfN(_FloatNx complex z);
_FloatN complex CMPLXFN(_FloatN x, _FloatN y);
_FloatN complex CMPLXFN(_FloatNx x, _FloatNx y);
```
In 7.31.1, change:

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

to:

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

15 Type-generic macros <tgmath.h>

The following changes to C11 + TS18661-1 + TS18661-2 enhance the specification of type-generic macros in <tgmath.h> to apply to interchange and extended floating types, as well as standard floating types.

Changes to C11 + TS18661-1 + TS18661-2:

In 7.25, replace paragraph #3b:

[3b] If arguments for generic parameters of a type-generic macro are such that some argument has a corresponding real type that is of standard floating type and another argument is of decimal floating type, the behavior is undefined.

with:

[3b] 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.

In 7.25#3c, replace the bullets:

— First, if any argument for generic parameters has type _Decimal128, the type determined is _Decimal128.

— Otherwise, if any argument for generic parameters has type _Decimal64, or if any argument for generic parameters is of integer type and another argument for generic parameters has type _Decimal32, the type determined is _Decimal64.

— Otherwise, if any argument for generic parameters has type _Decimal32, the type determined is _Decimal32.
— Otherwise, if the corresponding real type of any argument for generic parameters is long double, the type determined is long double.

— Otherwise, if the corresponding real type of any argument for generic parameters is double or is of integer type, the type determined is double.

— Otherwise, if any argument for generic parameters is of integer type, the type determined is double.

— Otherwise, the type determined is float.

with:

— If two arguments 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 any arguments for generic parameters have type _DecimalM where M ≥ 64 or _DecimalNx where N ≥ 32, the type determined is the widest of the types of these arguments. If _DecimalM and _DecimalNx are both widest types (with equivalent sets of values) of these arguments, the type determined is _DecimalM.

— Otherwise, if any argument for generic parameters is of integer type and another argument for generic parameters has type _Decimal32, the type determined is _Decimal64.

— Otherwise, if any argument for generic parameters has type _Decimal32, the type determined is _Decimal32.

— Otherwise, if the corresponding real type of any argument for generic parameters has type long double, _FloatM where M ≥ 128, or _FloatNx where N ≥ 64, the type determined is the widest of the corresponding real types of these arguments. If _FloatM and either long double or _FloatNx are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is _FloatM. Otherwise, if long double and _FloatNx are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is long double.

— Otherwise, if the corresponding real type of any argument for generic parameters has type double, _Float64, or _Float32x, the type determined is the widest of the corresponding real types of these arguments. If _Float64 and either double or _Float32x are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is _Float64. Otherwise, if double and _Float32x are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is double.

— Otherwise, if any argument for generic parameters is of integer type, the type determined is double.

— Otherwise, if the corresponding real type of any argument for generic parameters has type _Float32, the type determined is _Float32.

— Otherwise, the type determined is float.

In the second bullet 7.25#3c, attach a footnote to the wording:

the type determined is the widest
where the footnote is:

*) The term widest here refers to a type whose set of values is a superset of (or equivalent to) the sets of values of the other types.

In 7.25#6, replace:

Use of the macro with any argument of standard floating or complex type invokes a complex function. Use of the macro with an argument of decimal floating type results in undefined behavior.

with:

Use of the macro 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.

After 7.25#6c, add the paragraph:

[6d] For an implementation that provides the following real floating types:

type | IEC 60559 format
-----|------------------
float | binary32
double | binary64
long double | binary128
_Float32 | binary32
_Float64 | binary64
_Float32x | binary64
_Float64x | binary128

a type-generic macro `cbrt` that conforms to the specification in this clause and that is affected by constant rounding modes could be implemented as follows:

```c
#include <stdc_common_macros.h>

#if defined(__STDC_WANT_IEC_60559_TYPES_EXT__)
#define cbrt(X)__Generic((X),
  _Float128: cbrtf128(X),
  _Float64: cbrtf64(X),
  _Float32: cbrtf32(X),
  _Float64x: cbrtf64x(X),
  _Float32x: cbrtf32x(X),
  long double: cbrtl(X),
  default: _Roundwise_cbrt(X),
  float: cbrtf(X)
)
#else
#define cbrt(X)__Generic((X),
  long double: cbrtl(X),
  default: _Roundwise_cbrt(X),
  float: cbrtf(X)
)
#endif
```
where \texttt{Roundwise\_cbrt()} is equivalent to \texttt{cbrt()} invoked without macro-replacement suppression.

In 7.25#7, insert at the beginning of the example:

\begin{verbatim}
#define __STDC_WANT_IEC_60559_TYPES_EXT__
\end{verbatim}

In 7.25#7, append to the declarations:

\begin{verbatim}
#if __STDC_IEC_60559_TYPES__ >= 201506L
  _Float32x f32x;
  _Float64 f64;
  _Float128 f128;
  _Float64x complex f64xc;
#endif
\end{verbatim}

In 7.25#7, append to the table:

\begin{verbatim}
cos(f64xc)    ccosf64x(f64xc)
pow(dc, f128) cpowf128(dc, f128)
fmax(f64, d)  fmaxf64(f64, d)
fmax(d, f32x) 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)
pow(f32x, n)  powf32x(f32x, n)
\end{verbatim}
Bibliography


