ISO/IEC JTC 1/SC 22/WG 14 N1896

Date: yyyy-mm-dd

Reference number of document: **ISO/IEC TS 18661-3**

Committee identification: ISO/IEC JTC 1/SC 22/WG 14

Secretariat: ANSI

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

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Specification
Document subtype:
Document stage: (20) Preparation
Document language: E

Contents

Page

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*, 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*

— *Part 5: Supplementary attributes*

Part 1 updates ISO/IEC 9899:2011, *Information technology — Programming Language C*, Annex F in particular, to support all required features of ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*.

Part 2 supersedes ISO/IEC TR 24732:2009, *Information technology — Programming languages, their environments and system software interfaces — Extension for the programming language C to support decimal floating-point arithmetic*.

Parts 3-5 specify extensions to ISO/IEC 9899:2011 for features recommended in ISO/IEC/IEEE 60559:2011.

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:

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:

> *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 data that may be exchanged 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.

ISO/IEC 9899:2011 (C11) includes refinements to the C99 floating-point specification, though is still based on IEC 60559:1989. C11 upgrades Annex G from “informative” to “conditionally normative”.

ISO/IEC TR 24732:2009 introduced partial C support for the decimal floating-point arithmetic in ISO/IEC/IEEE 60559:2011. ISO/IEC TR 24732, for which technical content was completed while IEEE 754-2008 was still in the later stages of development, specifies decimal types based on ISO/IEC/IEEE 60559:2011 decimal formats, though it does not include all of the operations required by ISO/IEC/IEEE 60559:2011.

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.

Part 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 Part 1.

Part 2 enhances ISO/IEC TR 24732 to cover all the requirements, plus some basic recommendations, of ISO/IEC/IEEE 60559:2011 for decimal floating-point arithmetic. C implementations intending to provide an extension for decimal floating-point arithmetic supporting ISO/IEC/IEEE 60559:2011 are expected to conform to Part 2.

Part 3, this document, specifies types and other support for interchange and extended formats recommended in ISO/IEC/IEEE 60559:2011. C implementations intending to provide an extension for these formats are expected to conform to Part 3.

Part 4 specifies functions for operations recommended in ISO/IEC/IEEE 60559:2011. C implementations intending to provide an extension for these operations are expected to conform to Part 4.

Part 5 specifies support for attributes recommended in ISO/IEC/IEEE 60559:2011. C implementations intending to provide an extension for these attributes are expected to conform to Part 5.

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 ≥ 128 and a multiple of 32, and
  - for decimal, k ≥ 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, not their specific 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

— constants

— function suffixes

— character sequence conversion interfaces

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 **201**ymm**L**.

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 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*

ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*

ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic* (with identical content to IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronic Engineers, Inc., New York, 2008)

ISO/IEC 18661-1:2014, *Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 1: Binary floating-point arithmetic*

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

Changes specified in this part of ISO/IEC TS 18661 are relative to ISO/IEC 9899:2011, including *Technical Corrigendum 1* (ISO/IEC 9899:2011/Cor. 1:2012), together with the changes from parts 1 and 2 of ISO/IEC TS 18661.

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9899:2011, ISO/IEC/IEEE 60559:2011, and the following apply.

4.1
C11
standard ISO/IEC 9899:2011, *Information technology — Programming languages C*, including *Technical Corrigendum 1* (ISO/IEC 9899:2011/Cor. 1:2012)

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:

> **`__STDC_IEC_60559_TYPES__`** The integer constant **`201ymmL`**, 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 **`_Float`***N*:

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 **`_Float`***N***`x`**:

FLT*N*X_MANT_DIG	**FLT*N*X_MIN_10_EXP**	**FLT*N*X_EPSILON**
FLT*N*X_DECIMAL_DIG	**FLT*N*X_MAX_EXP**	**FLT*N*X_MIN**
FLT*N*X_DIG	**FLT*N*X_MAX_10_EXP**	**FLT*N*X_TRUE_MIN**
FLT*N*X_MIN_EXP	**FLT*N*X_MAX**	

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

DEC*N*_MANT_DIG	**DEC*N*_MAX**	**DEC*N*_TRUE_MIN**
DEC*N*_MIN_EXP	**DEC*N*_EPSILON**	
DEC*N*_MAX_EXP	**DEC*N*_MIN**	

for supported types **_Decimal*N*x**:

DEC*N*X_MANT_DIG	**DEC*N*X_MAX**	**DEC*N*X_TRUE_MIN**
DEC*N*X_MIN_EXP	**DEC*N*X_EPSILON**	
DEC*N*X_MAX_EXP	**DEC*N*X_MIN**	

After 7.3#2, insert the paragraph:

[2a] The following identifiers are declared or defined only if **__STDC_WANT_IEC_60559_TYPES_EXT__** is defined as a macro at the point in the source file where **<complex.h>** is first included:

for supported types **_Float***N*:

cacosf*N*	**catanhf***N*	**csqrtf***N*
casinf*N*	**ccoshf***N*	**cargf***N*
catanf*N*	**csinhf***N*	**cimagf***N*
ccosf*N*	**ctanhf***N*	**CMPLXF***N*
csinf*N*	**cexpf***N*	**conjf***N*
ctanf*N*	**clogf***N*	**cprojf***N*
cacoshf*N*	**cabsf***N*	**crealf***N*
casinhf*N*	**cpowf***N*	

for supported types **_Float*N*x**:

cacosf*N*x	**catanhf*N*x**	**csqrtf*N*x**
casinf*N*x	**ccoshf*N*x**	**cargf*N*x**
catanf*N*x	**csinhf*N*x**	**cimagf*N*x**
ccosf*N*x	**ctanhf*N*x**	**CMPLXF*N*X**
csinf*N*x	**cexpf*N*x**	**conjf*N*x**
ctanf*N*x	**clogf*N*x**	**cprojf*N*x**
cacoshf*N*x	**cabsf*N*x**	**crealf*N*x**
casinhf*N*x	**cpowf*N*x**	

After 7.12#1c, insert the paragraph:

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

long_double_t

for supported types **_Float***N*:

_Float*N*_t	**log1pf***N*	**fromfpf***N*
HUGE_VAL_F*N*	**log2f***N*	**ufromfpf***N*
SNANF*N*	**logbf***N*	**fromfpxf***N*
FP_FAST_FMAF*N*	**modff***N*	**ufromfpxf***N*
acosf*N*	**scalbnf***N*	**fmodf***N*
asinf*N*	**scalblnf***N*	**remainderf***N*

atanfN	cbrtfN	remquofN
atan2fN	fabsfN	copysignfN
cosfN	hypotfN	nanfN
sinfN	powfN	nextafterfN
tanfN	sqrtfN	nextupfN
acoshfN	erffN	nextdownfN
asinhfN	erfcfN	canonicalizefN
atanhfN	lgammafN	encodefN
coshfN	tgammafN	decodefN
sinhfN	ceilfN	fdimfN
tanhfN	floorfN	fmaxfN
expfN	nearbyintfN	fminfN
exp2fN	rintfN	fmaxmagfN
expm1fN	lrintfN	fminmagfN
frexpfN	llrintfN	fmafN
ilogbfN	roundfN	totalorderfN
ldexpfN	lroundfN	totalordermagfN
llogbfN	llroundfN	getpayloadfN
logfN	truncfN	setpayloadfN
log10fN	roundevenfN	setpayloadsigfN

for supported types _FloatNx:

HUGE_VAL_FNX	logbfNx	fromfpfNx
SNANFNX	modffNx	ufromfpfNx
FP_FAST_FMAFNX	scalbnfNx	fromfpxfNx
acosfNx	scalblnfNx	ufromfpxfNx
asinfNx	cbrtfNx	fmodfNx
atanfNx	fabsfNx	remainderfNx
atan2fNx	hypotfNx	remquofNx
cosfNx	powfNx	copysignfNx
sinfNx	sqrtfNx	nanfNx
tanfNx	erffNx	nextafterfNx
acoshfNx	erfcfNx	nextupfNx
asinhfNx	lgammafNx	nextdownfNx
atanhfNx	tgammafNx	canonicalizefNx
expfNx	ceilfNx	fdimfNx
exp2fNx	floorfNx	fmaxfNx
expm1fNx	nearbyintfNx	fminfNx
frexpfNx	rintfNx	fmaxmagfNx
ilogbfNx	lrintfNx	fminmagfNx
llogbfNx	llrintfNx	fmafNx
ldexpfNx	roundfNx	totalorderfNx
logfNx	lroundfNx	totalordermagfNx
log10fNx	llroundfNx	getpayloadfNx
log1pfNx	truncfNx	setpayloadfNx
log2fNx	roundevenfNx	setpayloadsigfNx

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

FP_FAST_FMADDFN	FP_FAST_FMFMAFN	fMmulfN
FP_FAST_FMSUBFN	FP_FAST_FMSQRTFN	fMdivfN
FP_FAST_FMMULFN	fMaddfN	fMfmafN
FP_FAST_FMDIVFN	fMsubfN	fMsqrtfN

for supported types **_Float***M* and **_Float***N***x** where $M \le N$:

FP_FAST_F*M***ADDF***N***X**	**FP_FAST_F***M***FMAF***N***X**	**f***M***mulf***N***x**
FP_FAST_F*M***SUBF***N***X**	**FP_FAST_F***M***SQRTF***N***X**	**f***M***divf***N***x**
FP_FAST_F*M***MULF***N***X**	**f***M***addf***N***x**	**f***M***fmaf***N***x**
FP_FAST_F*M***DIVF***N***X**	**f***M***subf***N***x**	**f***M***sqrtf***N***x**

for supported types **_Float***M***x** and **_Float***N* where $M < N$:

FP_FAST_F*M***XADDF***N*	**FP_FAST_F***M***XFMAF***N*	**f***M***xmulf***N*
FP_FAST_F*M***XSUBF***N*	**FP_FAST_F***M***XSQRTF***N*	**f***M***xdivf***N*
FP_FAST_F*M***XMULF***N*	**f***M***xaddf***N*	**f***M***xfmaf***N*
FP_FAST_F*M***XDIVF***N*	**f***M***xsubf***N*	**f***M***xsqrtf***N*

for supported types **_Float***M***x** and **_Float***N***x** where $M < N$:

FP_FAST_F*M***XADDF***N***X**	**FP_FAST_F***M***XFMAF***N***X**	**f***M***xmulf***N***x**
FP_FAST_F*M***XSUBF***N***X**	**FP_FAST_F***M***XSQRTF***N***X**	**f***M***xdivf***N***x**
FP_FAST_F*M***XMULF***N***X**	**f***M***xaddf***N***x**	**f***M***xfmaf***N***x**
FP_FAST_F*M***XDIVF***N***X**	**f***M***xsubf***N***x**	**f***M***xsqrtf***N***x**

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

f*M***encf***N*

for supported types **_Decimal***N*, where $N \ne 32$, 64, and 128:

_Decimal*N***_t**	**logbd***N*	**fmodd***N*
HUGE_VAL_D*N*	**modfd***N*	**remainderd***N*
SNAND*N*	**scalbnd***N*	**copysignd***N*
FP_FAST_FMAD*N*	**scalblnd***N*	**nand***N*
acosd*N*	**cbrtd***N*	**nextafterd***N*
asind*N*	**fabsd***N*	**nextupd***N*
atand*N*	**hypotd***N*	**nextdownd***N*
atan2d*N*	**powd***N*	**canonicalized***N*
cosd*N*	**sqrtd***N*	**quantized***N*
sind*N*	**erfd***N*	**samequantumd***N*
tand*N*	**erfcd***N*	**quantumd***N*
acoshd*N*	**lgammad***N*	**llquantexpd***N*
asinhd*N*	**tgammad***N*	**encodedecd***N*
atanhd*N*	**ceild***N*	**decodedecd***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*	**fmind***N*
expm1d*N*	**roundd***N*	**fmaxmagd***N*
frexpd*N*	**lroundd***N*	**fminmagd***N*
ilogbd*N*	**llroundd***N*	**fmad***N*
llogbd*N*	**truncd***N*	**totalorderd***N*
ldexpd*N*	**roundevend***N*	**totalordermagd***N*
logd*N*	**fromfpd***N*	**getpayloadd***N*
log10d*N*	**ufromfpd***N*	**setpayloadd***N*
log1pd*N*	**fromfpxd***N*	**setpayloadsigd***N*
log2d*N*	**ufromfpxd***N*	

for supported types _Decimal*N*x:

HUGE_VAL_D*N*X	log2d*N*x	ufromfpd*N*x
SNAND*N*X	logbd*N*x	fromfpxd*N*x
FP_FAST_FMAD*N*X	modfd*N*x	ufromfpxd*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	copysignd*N*x
atan2d*N*x	fabsd*N*x	nand*N*x
cosd*N*x	hypotd*N*x	nextafterd*N*x
sind*N*x	powd*N*x	nextupd*N*x
tand*N*x	sqrtd*N*x	nextdownd*N*x
acoshd*N*x	erfd*N*x	canonicalized*N*x
asinhd*N*x	erfcd*N*x	quantized*N*x
atanhd*N*x	lgammad*N*x	samequantumd*N*x
coshd*N*x	tgammad*N*x	quantumd*N*x
sinhd*N*x	ceild*N*x	llquantexpd*N*x
tanhd*N*x	floord*N*x	fdimd*N*x
expd*N*x	nearbyintd*N*x	fmaxd*N*x
exp2d*N*x	rintd*N*x	fmind*N*x
expm1d*N*x	lrintd*N*x	fmaxmagd*N*x
frexpd*N*x	llrintd*N*x	fminmagd*N*x
ilogbd*N*x	roundd*N*x	fmad*N*x
llogbd*N*x	lroundd*N*x	totalorderd*N*x
ldexpd*N*x	llroundd*N*x	totalordermagd*N*x
logd*N*x	truncd*N*x	getpayloadd*N*x
log10d*N*x	roundevend*N*x	setpayloadd*N*x
log1pd*N*x	fromfpd*N*x	setpayloadsigd*N*x

for supported types **_Decimal***M* and **_Decimal***N* where $M < N$ and M and N are not both one of 32, 64, and 128:

FP_FAST_D*M*ADDD*N*	FP_FAST_D*M*FMAD*N*	d*M*muld*N*
FP_FAST_D*M*SUBD*N*	FP_FAST_D*M*SQRTD*N*	d*M*divd*N*
FP_FAST_D*M*MULD*N*	d*M*addd*N*	d*M*fmad*N*
FP_FAST_D*M*DIVD*N*	d*M*subd*N*	d*M*sqrtd*N*

for supported types **_Decimal***M* and **_Decimal***N***x** where $M \le N$:

FP_FAST_D*M*ADDD*N*X	FP_FAST_D*M*FMAD*N*X	d*M*muld*N*x
FP_FAST_D*M*SUBD*N*X	FP_FAST_D*M*SQRTD*N*X	d*M*divd*N*x
FP_FAST_D*M*MULD*N*X	d*M*addd*N*x	d*M*fmad*N*x
FP_FAST_D*M*DIVD*N*X	d*M*subd*N*x	d*M*sqrtd*N*x

for supported types **_Decimal***M***x** and **_Decimal***N* where $M < N$:

FP_FAST_D*M*XADDD*N*	FP_FAST_D*M*XFMAD*N*	d*M*xmuld*N*
FP_FAST_D*M*XSUBD*N*	FP_FAST_D*M*XSQRTD*N*	d*M*xdivd*N*
FP_FAST_D*M*XMULD*N*	d*M*xaddd*N*	d*M*xfmad*N*
FP_FAST_D*M*XDIVD*N*	d*M*xsubd*N*	d*M*xsqrtd*N*

for supported types **_Decimal***M***x** and **_Decimal***N***x** where *M* < *N*:

FP_FAST_D*M***XADDD***N***X**	**FP_FAST_D***M***XFMAD***N***X**	**d***M***xmuld***N***x**
FP_FAST_D*M***XSUBD***N***X**	**FP_FAST_D***M***XSQRTD***N***X**	**d***M***xdivd***N***x**
FP_FAST_D*M***XMULD***N***X**	**d***M***xaddd***N***x**	**d***M***xfmad***N***x**
FP_FAST_D*M***XDIVD***N***X**	**d***M***xsubd***N***x**	**d***M***xsqrtd***N***x**

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

d*M***encdecd***N*   **d***M***encbind***N*

After 7.22#1b, insert the paragraph:

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

for supported types **_Float***N*:

strfromf*N*   **strtof***N*

for supported types **_Float***N***x**:

strfromf*N***x**   **strtof***N***x**

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

strfromd*N*   **strtod***N*

for supported types **_Decimal***N***x**:

strfromd*N***x**   **strtod***N***x**

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

strfromencf*N*   **strtoencf***N*

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

strfromencdecd*N*   **strtoencdecd***N*
strfromencbind*N*   **strtoencbind***N*

6 Types

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

_Float*N* for binary interchange formats

_Decimal*N* for decimal interchange formats

_Float*N***x** for binary extended formats

_Decimal*N***x** 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 *standard floating types* as a collective name for the types **`float`**, **`double`**, and **`long double`** and it defined *decimal floating types* as a collective name for the types **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`**. This part of ISO/IEC TS 18661 extends the definition of decimal floating types and defines *binary floating types* to be collective names for types for all the appropriate IEC 60559 arithmetic formats. Thus real floating types are classified as follows:

standard floating types:
float
double
long double

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

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

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 **`_Decimal32`***), **`_Decimal64`**, and **`_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

Parameter	binary16	binary32	binary64	binary128	binary*N* ($N \geq 128$)
N, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$N - \text{round}(4\times\log_2(N)) + 13$
emax, maximum exponent *e*	15	127	1023	16383	$2^{(N-p-1)} - 1$

Encoding parameters					
bias, *E*−*e*	15	127	1023	16383	*emax*
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	round(4×$\log_2$(*N*)) − 13
t, trailing significand field width in bits	10	23	52	112	*N* − *w* − 1
N, storage width in bits	16	32	64	128	1 + *w* + *t*

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

Decimal interchange format parameters

Parameter	decimal32	decimal64	decimal128	decimal*N* (*N* ≥ 32)
N, storage width in bits	32	64	128	multiple of 32
p, precision in digits	7	16	34	9 × *N*/32 − 2
emax, maximum exponent *e*	96	384	6144	$3 \times 2^{(N/16 + 3)}$
Encoding parameters				
bias, *E*−*e*	101	398	6176	*emax* + *p* − 2
sign bit	1	1	1	1
W+5, combination field width in bits	11	13	17	*N*/16 + 9
t, trailing significand field width in bits	20	50	110	15×*N*/16 − 10
N, storage width in bits	32	64	128	1 + 5 + *w* + *t*

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

[10b] Types designated

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

and types designated

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

are collectively called the *interchange floating types*. Each interchange floating type has the IEC 60559 interchange format corresponding to its width (*N*) and radix (2 for `_Float`*N*, 10 for `_Decimal`*N*). 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 `_Float`*N* as an interchange floating type with the same representation and alignment requirements as `long double`. The implementation may provide other binary interchange floating types; the set of such types supported is implementation-defined.

[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 (7.12.11.7c), 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

	Extended formats associated with:				
Parameter	binary32	binary64	binary128	decimal64	decimal128
p digits ≥	32	64	128	22	40
emax ≥	1023	16383	65535	6144	24576

[10g] Types designated **`_Float32x`**, **`_Float64x`**, **`_Float128x`**, **`_Decimal64x`**, 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: **`_Decimal64x`**, which may have the same set of values as **`_Decimal128`**, and may provide **`_Decimal128x`**. Which (if any) of the optional extended floating types are provided is implementation-defined.

[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 **`_Float`***N* and the extended floating types designated **`_Float`***N***`x`** are collectively called the *binary floating types*. The interchange floating types designated **`_Decimal`***N* and the extended floating types designated **`_Decimal`***N***`x`** 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`**.43) (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 **_Float***N*, and the extended floating types **_Float***N***x**, there are *complex types* designated respectively as **float _Complex**, **double _Complex**, **long double _Complex**, **_Float***N* **_Complex**, and **_Float***N***x _Complex**. 43) (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:

_Decimal32
_Decimal64
_Decimal128

with:

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

In the list of type specifiers in 6.7.2, replace:

_Decimal32
_Decimal64
_Decimal128

with:

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

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

— **_Decimal32**

— **_Decimal64**

— **_Decimal128**

with:

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

— **_Float32x**

— **_Float64x**

— **_Float128x**

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

— **`_Decimal64x`**

— **`_Decimal128x`**

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

— **`_Float32x _Complex`**

— **`_Float64x _Complex`**

— **`_Float128x _Complex`**

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 **`_Float`***N* (where *N* is 16, 32, 64, or ≥ 128 and a multiple of 32), **`_Float32x`**, **`_Float64x`**, **`_Float128x`**, **`_Decimal`***N* (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 **`_Float`***N*, and the extended floating types **`_Float`***N***`x`**, there are *imaginary types* designated respectively as **`float _Imaginary`**, **`double _Imaginary`**, **`long double _Imaginary`**, **`_Float`***N* **`_Imaginary`**, and **`_Float`***N***`x _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   0.000001E-95DF
DEC64_TRUE_MIN   0.000000000000001E-383DD
DEC128_TRUE_MIN  0.000000000000000000000000000000001E-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***N***X_** 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***N***X_** 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***N* is a supported interchange floating type
: evaluate operations and constants, whose semantic type has at most the range and precision of the **_Float***N* type, to the range and precision of the **_Float***N* type; evaluate all other operations and constants to the range and precision of the semantic type;

N + 1, where **_Float***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***N***x** type, to the range and precision of the **_Float***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 **_Decimal***N* is a supported interchange floating type
: evaluate operations and constants, whose semantic type has at most the range and precision of the **_Decimal***N* type, to the range and precision of the **_Decimal***N* type; evaluate all other operations and constants to the range and precision of the semantic type;

N + 1, where **_Decimal***N***x** is a supported extended floating type
: evaluate operations and constants, whose semantic type has at most the range and precision of the **_Decimal***N***x** type, to the range and precision of the **_Decimal***N***x** 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 **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*

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

— number of digits in the coefficient, *p*

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

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

FLT*N*_DECIMAL_DIG
FLT*N*X_DECIMAL_DIG

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

FLT*N*_DIG
FLT*N*X_DIG

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

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

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

FLT*N*_MIN_10_EXP
FLT*N*X_MIN_10_EXP

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

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

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

FLT*N*_MAX_10_EXP
FLT*N*X_MAX_10_EXP

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

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

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

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

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

```
FLTN_MIN
FLTNX_MIN
DECN_MIN
DECNX_MIN
```

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

```
FLTN_TRUE_MIN
FLTNX_TRUE_MIN
DECN_TRUE_MIN
DECNX_TRUE_MIN
```

With the following change, **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.226)

to:

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, f***N*, **f***N***x**, **d***N*, and **d***N***x** suffixes, which are corresponding functions whose parameters, return values, or both are of types **`float, long double, _Float`***N*, **`_Float`***N***`x`**, **`_Decimal`***N*, and **`_Decimal`***N***`x`**, respectively.226)

Add after 7.12#1d:

> [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.227)

to:

> [2] The types
>
> ```
> float_t
> double_t
> long_double_t
> ```
>
> and for each supported type `_Float`*N*, the type
>
> `_Float`*N*`_t`
>
> and for each supported type `_Decimal`*N*, the type
>
> `_Decimal`*N*`_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`, `_Float`*N*, and `_Decimal`*N*, 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`;
>
> — `_Float`*N*`_t` has at least the range and precision of `_Float`*M*`_t` if $N > M$;
>
> — `_Decimal`*N*`_t` has at least the range and precision of `_Decimal`*M*`_t` if $N > M$.
>
> If `FLT_RADIX` is 2 and `FLT_EVAL_METHOD` 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
>
> **HUGE_VAL_F***N*
> **HUGE_VAL_D***N*
> **HUGE_VAL_F***N***X**
> **HUGE_VAL_D***N***X**
>
> expand to constant expressions of types **_Float***N*, **_Decimal***N*, **_Float***N***x**, and **_Decimal***N***x**, respectively, representing positive infinity.

Replace 7.12#5b:

> [5b] The decimal signaling NaN macros
>
> **SNAND32**
> **SNAND64**
> **SNAND128**
>
> 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
>
> **SNANF***N*
> **SNAND***N*
> **SNANF***N***X**
> **SNAND***N***X**

expand to constant expressions of types **_Float***N*, **_Decimal***N*, **_Float***N***x**, and **_Decimal***N***x**, respectively, representing a signaling NaN. If a signaling NaN macro is used for initializing an object of the same type that has static or thread-local storage duration, the object is initialized with a signaling NaN value.

Replace 7.12#7b:

[7b] The macros

```
FP_FAST_FMAD32
FP_FAST_FMAD64
FP_FAST_FMAD128
```

are, respectively, **_Decimal32**, **_Decimal64**, and **_Decimal128** analogues of **FP_FAST_FMA**.

with:

[7b] The macros

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

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

Replace 7.12#7c:

[7c] The macros

```
FP_FAST_D32ADDD64
FP_FAST_D32ADDD128
FP_FAST_D64ADDD128
FP_FAST_D32SUBD64
FP_FAST_D32SUBD128
FP_FAST_D64SUBD128
FP_FAST_D32MULD64
FP_FAST_D32MULD128
FP_FAST_D64MULD128
FP_FAST_D32DIVD64
FP_FAST_D32DIVD128
FP_FAST_D64DIVD128
FP_FAST_D32FMAD64
FP_FAST_D32FMAD128
FP_FAST_D64FMAD128
FP_FAST_D32SQRTD64
FP_FAST_D32SQRTD128
FP_FAST_D64SQRTD128
```

are decimal analogues of **FP_FAST_FADD**, **FP_FAST_FADDL**, **FP_FAST_DADDL**, etc.

with:

[7c] The macros in the following lists are interchange and extended floating type analogues of **FP_FAST_FADD**, **FP_FAST_FADDL**, **FP_FAST_DADDL**, etc.

[7d] For *M* < *N*, the macros

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

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

[7e] For *M* ≤ *N*, the macros

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

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

[7f] For *M* < *N*, the macros

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

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

[7g] For *M* < *N*, the macros

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

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

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 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>`** functon 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>`** functon listed in the table above indicates the family of functions of all standard and binary floating types (for example, **`acosf`**, **`acosl`**, **`acosf`***N*, and **`acosf`***N***`x`** as well as **`acos`**).

After 7.6.1a#4, add:

> [4a] The **`f`***M***`encf`***N*, **`strfromencf`***N*, and **`strtoencf`***N* functions for binary interchange types are also affected by constant rounding modes.

In 7.6.1b#2 after the table, add:

> Each **`<math.h>`** functon listed in the table above indicates the family of functions of all decimal floating types (for example, **`acosd`***N***`x`**, as well as **`acosd`***N*).

After 7.6.1b#2, add:

> [3] The **`d`***M***`encbind`***N*, **`d`***M***`encdecd`***N*, **`strfromencbind`***N*, **`strfromencdecd`***N*, **`strtoencbind`***N*, and **`strtoencdecd`***N* functions for decimal interchange types are also affected by constant rounding modes.

Change 7.6.3 from:

> The **`fegetround`** and **`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 **`fegetround`** function gets the current value of the dynamic rounding direction mode.

to:

> The **`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 **`fesetround`** function sets the dynamic rounding direction mode to the rounding direction represented by its argument **round**.

to:

> The **`fesetround`** function sets the dynamic rounding direction mode to the rounding direction represented by its argument **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 `d`*N* suffix is added, remove any prototypes with a `d32`, `d64`, or `d128` suffix.

7.12.4 Trigonometric functions

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

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

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

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

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

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

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

7.12.5 Hyperbolic functions

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

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



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

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

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

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

7.12.6 Exponential and logarithmic functions

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

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

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

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

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

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

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

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

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

_FloatN 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);
```

7.12.7 Power and absolute-value functions

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

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



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

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

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

7.12.8 Error and gamma functions

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

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

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

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

7.12.9 Nearest integer functions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.12.10 Remainder functions

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

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

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

7.12.11 Manipulation functions

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

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

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

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

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

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

_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);
```

7.12.12 Maximum, minimum, and positive difference functions

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

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

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

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

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

7.12.13 Floating multiply-add

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

7.12.14 Functions that round result to narrower format

```
_FloatM fMaddfN(_FloatN x, _FloatN y); // M < N
_FloatM fMaddfNx(_FloatNx x, _FloatNx y); // M <= N
_FloatMx fMxaddfN(_FloatN x, _FloatN y);  // M < N
_FloatMx fMxaddfNx(_FloatNx x, _FloatNx y);  // M < N
```

```
_DecimalM dMadddN(_DecimalN x, _DecimalN y); // M < N
_DecimalM dMadddNx(_DecimalNx x, _DecimalNx y);  // M <= N
_DecimalMx dMxadddN(_DecimalN x, _DecimalN y);   // M < N
_DecimalMx dMxadddNx(_DecimalNx x, _DecimalNx y);   // M < N

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

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

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

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

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

F.10.12 Total order functions

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

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

F.10.13 Payload functions

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

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

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

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** × $2^{*\mathbf{exp}}$, 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** × $2^{*\mathbf{exp}}$, 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** × $2^{\mathbf{exp}}$ when the type of the function is a standard floating type, or return **x** × $10^{\mathbf{exp}}$ when the type of the function is a decimal floating type.

to:

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

$$1 \le \mathbf{x} \times b^{-\mathbf{logb(x)}} < b$$

where b = **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**,

$$1 \le \mathbf{x} \times b^{-\mathbf{logb(x)}} < b$$

where b = **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** × b^{n}, where b = **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** × b^{n}, where b = **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 `encodef`*N* functions

Synopsis

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

Description

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

Returns

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

7.12.11.7b The `decodef`*N* functions

Synopsis

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

Description

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

Returns

[3] The **`decodef`***N* functions return no value.

7.12.11.7c Encoding-to-encoding conversion functions

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

7.12.11.7c.1 The **fMencfN** functions

Synopsis

[1]
```
#include <math.h>
void fMencfN(unsigned char * restrict encMptr, const unsigned char *
   restrict encNptr);
```

Description

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

Returns

[3] These functions return no value.

7.12.11.7c.2 The dMencdecdN and dMencbindN functions

Synopsis

[1]
```
#include <math.h>
void dMencdecdN(unsigned char * restrict encMptr, const unsigned char
   * restrict encNptr);
void dMencbindN(unsigned char * restrict encMptr, const unsigned char
   * restrict encNptr);
```

Description

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

Returns

[3] These functions return no value.

In F.3, change the row:

convertFormat – different formats	cast and implicit conversions	6.3.1.5, 6.5.4

to:

convertFormat – different formats	cast, implicit conversions, conversion functions (details below)	6.3.1.5, 6.5.4, 7.12.11, 7.22.11b, 7.22.1 (details below)

After F.3 [3], add:

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

- For conversions between arithmetic formats supported by floating types - casts and implicit conversions.
- For same-radix conversions between non-arithmetic interchange formats - encoding-to-encoding conversion functions (7.12.11.7c).
- For conversions between non-arithmetic interchange formats (same or different radix) – compositions of string-from-encoding functions (7.22.1.3c) (converting exactly) and string-to-encoding functions (7.22.1.3b).
- For same-radix conversions from interchange formats supported by interchange floating types to non-arithmetic interchange formats – compositions of **`encode`** functions (7.12.11.7a, 7.12.11b.1, 7.12.11b.3) and encoding-to-encoding (7.12.11.7c) functions.
- For same radix conversions from non-arithmetic interchange formats to interchange formats supported by interchange floating types – compositions of encoding-to-encoding conversion functions (7.12.11.7c) and **`decode`** functions (7.12.11.7b, 7.12.11b.2, 7.12.11b.4).
- For conversions from non-arithmetic interchange formats to arithmetic formats supported by floating types (same or different radix) – compositions of string-from-encoding functions (7.22.1.3c) (converting exactly) and **`strto`** functions (7.22.1.3, 7.22.1.3a).
- For conversions from arithmetic formats supported by floating types to non-arthmetic interchange formats (same or different radix) – compositions of **`strfrom`** functions (7.22.1.2a) (converting exactly) and string-to-encoding functions (7.22.1.3b).

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 **`strfromd`***N* 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 **`format`** string results in undefined behavior.

Returns

[3] The **`strfromd`***N* 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`**.

with:

7.22.1.2a The `strfromd`, `strfromf`, `strfroml`, `strfromfN`, `strfromfNx`, `strfromdN`, and `strfromdNx` functions

Synopsis

[1]
```
#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);
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);
```

Description

[2] The **strfromd**, **strfromf**, **strfroml**, **strfromf*N***, **strfromf*N*x**, **strfromd*N***, and **strfromd*N*x** 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**, **long double**, **_Float*N***, **_Float*N*x**, **_Decimal*N***, or **_Decimal*N*x**) 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] These 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**.

Change the first part of 7.22.1.3:

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

Synopsis

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

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`, `strtof`*N*, and `strtof`*N*`x` functions

Synopsis

[1]
```
#include <stdlib.h>
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**, **strtof***N*, and **strtof***N***x** functions convert the initial portion of the string pointed to by **nptr** to **double**, **float**, **long double**, **_Float***N*, and **_Float***N***x** 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, **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 **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned

to:

plus or minus **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VALF***N*, or **HUGE_VALF***N***X** is returned

In 7.22.1.3a, change:

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_DFP_EXT__
#include <stdlib.h>
_Decimal32 strtod32(const char * restrict nptr, char ** restrict
   endptr);
_Decimal64 strtod64(const char * restrict nptr, char ** restrict
   endptr);
_Decimal128 strtod128(const char * restrict nptr, char ** restrict
   endptr);
```

to:

Synopsis

[1]
```
#include <stdlib.h>
_DecimalN strtodN(const char * restrict nptr, char ** restrict
   endptr);
_DecimalNx strtodNx(const char * restrict nptr, char ** restrict
   endptr);
```

After 7.22.1.3a, insert:

7.22.1.3b String-to-encoding functions

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

7.22.1.3b.1 The strtoencf*N* functions

Synopsis

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

Description

[2] The **`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 `strfromencf`*N* functions

Synopsis

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

Description

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

Returns

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

7.22.1.3c.2 The `strfromencdecd`*N* and `strfromencbind`*N* functions

Synopsis

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


Description

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

Returns

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

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**, **f***N*, and **f***N***x** suffixes which are corresponding functions whose parameters and return values have corresponding real types **float**, **long double**, **_Float***N*, and **_Float***N***x**.

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

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

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

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

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

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

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

7.3.7 Exponential and logarithmic functions

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

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

7.3.8 Power and absolute value functions

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

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

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

7.3.9 Manipulation functions

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

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

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

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

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

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

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**, **f***N*, or **f***N***x** 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 `_Decimal`*M* where $M \geq 64$ or `_Decimal`*N*`x` where $N \geq 32$, the type determined is the widest of the types of these arguments. If `_Decimal`*M* and `_Decimal`*N*`x` are both widest types (with equivalent sets of values) of these arguments, the type determined is `_Decimal`*M*.

— 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`, `_Float`*M* where $M \geq 128$, or `_Float`*N*`x` where $N \geq 64$, the type determined is the widest of the corresponding real types of these arguments. If `_Float`*M* and either `long double` or `_Float`*N*`x` are both widest corresponding real types (with equivalent sets of values) of these arguments, the type determined is `_Float`*M*. Otherwise, if `long double` and `_Float`*N*`x` 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
_Float128	binary128
_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:

```
#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 **_Roundwise_cbrt()** is equivalent to **cbrt()** invoked without macro-replacement suppression.

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

```
#define __STDC_WANT_IEC_60559_TYPES_EXT__
```

In 7.25#7, append to the declarations:

```
#if __STDC_IEC_60559_TYPES__ >= 201ymmL
    _Float32x f32x;
    _Float64 f64;
    _Float128 f128;
    _Float64x complex f64xc;
#endif
```

In 7.25#7, append to the table:

`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)`

Bibliography

[1] ISO/IEC TR 24732:2009, *Information technology – Programming languages, their environments and system software interfaces – Extension for the programming language C to support decimal floating-point arithmetic*

[2] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition*

[3] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

[4] IEEE 754−1985, *IEEE Standard for Binary Floating-Point Arithmetic*

[5] IEEE 854−1987, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*