ISO/IEC JTC 1/SC 22/WG 14 N1691

Date: yyyy-mm-dd

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

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 prolongée

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) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TS 18661 was prepared by Technical Committee ISO JTC 1, *Information Technology*, Subcommittee SC 22, *Programming languages, their environments, and system software interfaces*.

ISO/IEC TS 18661 consists of the following parts, under the general title *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: Supplemental functions*

— *Part 5: Supplemental attributes*

Part 1 updates ISO/IEC 9899:2011 (*Information technology — Programming languages, their environments and system software interfaces — 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 (It 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 IEC 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 inquiries 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 it 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 Technical Report 24732:2009 introduced partial C support for the decimal floating-point arithmetic in IEC 60559:2011. 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 IEC 60559:2011 decimal formats, though it does not include all of the operations required by IEC 60559:2011.

Purpose

The purpose of this Technical Specification is to provide a C language binding for IEC 60559:2011, based on the C11 standard, that delivers the goals of IEC 60559 to users and is feasible to implement. It is organized into five Parts.

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

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

Part 3 (this document), Part 4 (Supplementary functions), and Part 5 (Supplementary attributes) cover recommended features of IEC 60559-2011. C implementations intending to provide extensions for these features are expected to conform to the corresponding Parts.

Additional background on formats

The 2011 revision of the ISO/IEC 60559 standard for floating-point arithmetic 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 this Technical Specification.

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 Part 1 of Technical Specification 18661) signifying that it supports Annex F of the C Standard, then its **`float`** and **`double`** formats must be IEC 60559 binary32 and binary64.

Part 2 of Technical Specification 18661 defines types **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** with IEC 60559 formats decimal32, decimal64, and decimal128. Although IEC 60559 regards decimal32 as an interchange format, not a basic format, and does not require decimal32 arithmetic (other than conversions), Part 2 of Technical Specification 18661 has full arithmetic and library support for **`_Decimal32`**, just like for **`_Decimal64`** and **`_Decimal128`**.

The C language provides just three "generic" floating types (**`float`**, **`double`**, and **`long double`**), which Annex F of the C Standard requires 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.

Part 3 of Technical Specification 18661, this document, provides nomenclatures for types with IEC 60559 interchange and extended formats that allow portable use of the formats as envisioned in IEC 60559. It covers these aspects of types with IEC 60559 interchange and extended formats:

- names
- characteristics
- conversions
- constants
- function suffixes
- character sequence conversion interfaces

This specification includes interchange and extended nomenclatures for types that, in some cases, already have C nomenclatures. For example, a type with the IEC 60559 double format may be referred to as **`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 interchange and extended formats and C generic floating types.

Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 3: Interchange and extended types

1 Scope

This document, Part 3 of Technical Specification 18661, extends programming language C to include nomenclature for types with the interchange and extended floating-point formats specified in ISO/IEC/IEEE 60559:2011.

This document proposes nomenclature for all applicable types in Parts 1 and 2 of Technical Specification 18661 and for any other types with IEC 60559 interchange or extended formats supported by the implementation.

2 Conformance

An implementation conforms to Part 3 of Technical Specification 18661 if

a) It conforms for Part 1 or Part 2 (or both) of Technical Specification 18661;

b) It meets the requirements for a conforming implementation of C11 with all the suggested changes to C11 as specified in Part 3 of Technical Specification 18661; and

c) It defines **__STDC_IEC_60559_TYPES__** to **201**ymm**L.**

3 Normative references

The following referenced documents are indispensable for the application of this document. Only the editions cited apply.

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

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:yyyy, *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*

Suggested changes proposed by Part 3 of Technical Specification 18661 are relative to ISO/IEC 9899:2011 (C11). The actual specification is given by a synthesis with the suggested changes from Parts 1 or 2 or both, depending on which Parts the implementation supports.

ISSUE 1: Will the approach of the previous paragraph lead to a specification that is intelligible? Is there a better approach?

4 Terms and definitions

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

4.1
C11
standard ISO/IEC 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*

5 C standard conformance

5.1 Freestanding implementations

The following suggested change to C11 expands the conformance requirements for freestanding implements so that they might conform to this Part of Technical Specification18661

Suggested change to C11:

Replace the third sentence of 4#6:

> A *conforming freestanding implementation* shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **<float.h>**, **<iso646.h>**, **<limits.h>**, **<stdalign.h>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, **<stdint.h>**, and **<stdnoreturn.h>**.

with:

> A *conforming freestanding implementation* shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **<fenv.h>**, **<float.h>**, **<iso646.h>**, **<limits.h>**, **<math.h>**, **<stdalign.h>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, **<stdint.h>**, and **<stdnoreturn.h>** and the numeric conversion functions (7.22.1) of the standard header **<stdlib.h>**.

5.2 Predefined macros

Suggested change to C11:

In 6.10.8.3#1, add:

> **__STDC_IEC_60559_TYPES__** The integer constant **201ymmL**, intended to indicate support of interchange and extended types according to IEC 60559.

5.3 Standard headers

The library functions, macros, and types defined in this Part of Technical Specification 18661 are defined by their respective headers if the macro **__STDC_WANT_IEC_18661_EXT3__** is defined at the point in the source file where the appropriate header is first included.

6 Types

This clause recommends changes to C11 to include the interchange and extended types specified in IEC 60559.

Suggested change to C11:

Change the first sentence of 6.2.5#10 from:

[10] There are three *real floating types,* designated as **`float`**, **`double`**, and **`long double`**

to:

[10] There are three *generic floating types*, designated as **`float`**, **`double`**, and **`long double`**.

After 6.2.5#10, insert:

[10a] IEC 60559 specifies interchange formats, identified by their width, which can be used for the exchange of floating−point data between implementations. Tables 1 and 2 give parameters for the IEC 60559 interchange formats.

Table 1 – Binary interchange format parameters

Parameter	binary16	binary32	binary64	binary128	binary*N* (*N* ≥ 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	$\text{round}(4\times\log_2(N)) - 13$
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 Table 1 rounds to the nearest integer. For example, binary256 would have *p* = 237 and *emax* = 262143.

Table 2 – 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	$9 \times N/32 - 2$
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, exponent field width in bits	11	13	17	$N/16 + 9$
t, trailing significand field width in bits	20	50	110	$15\times N/16 - 10$
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

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

support the corresponding IEC 60559 interchange formats and are collectively called the *data-interchange types*. Each data-interchange type has the IEC 60559 interchange format corresponding to its width and radix. Data-interchange types that are supported by all applicable floating-point operations are collectively called the *interchange floating types*. Data-interchange types (including interchange floating types) are not compatible with any other types.

[10c] An implementation that defines **__STDC_IEC_60559_BFP__** shall provide

_Float32 and **_Float64** as interchange floating types with the same representation and alignment requirements as **float** and **double**, respectively, and

_Float16 as a data-interchange type.

If the implementation's **long double** type has an IEC 60559 interchange format of width *N*, 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**.

[10d] An implementation may provide any of the data-interchange types and may provide any of its data-interchange types as interchange floating types. For example, an implementation that defines **__STDC_IEC_60559_BFP__** may provide **_Float16** as an interchange floating type.

[10e] For each of its basic formats, IEC 60559 specifies as extended format whose maximum exponent and precision exceed those of the basic format it is associated with. Table 3 below gives the minimum values of these parameters:

Table 3 – 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

[10f] 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__** 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__** shall provide: **_Decimal64x**, which may have the same set of values as **_Decimal128**, and may provide **_Decimal128x**.

[10g] The generic floating types, interchange floating types, and extended floating types are collectively called the *real floating types*.

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 generic 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 6.2.5#14, change the first sentence from:

[14] The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. …

to:

[14] The type **char**, the signed and unsigned integer types, the floating types, and the data-interchange types are collectively called the *basic types*. …

In 6.2.5#21, change the first sentence from:

[21] Arithmetic types and pointer types are collectively called *scalar types*. …

to:

[21] Arithmetic types, data-interchange types, and pointer types are collectively called *scalar types*. …

Add the following to 6.4.1 Keywords:

keyword:
_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

Add the following to 6.7.2 Type specifiers:

type-specifier:
_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

Add the following bullets in 6.7.2#2 Constraints:

— **_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**

Add the following after 6.7.2#3:

> [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 interchange and extended types (see 6.10.8.3).

Add the following after 6.5#8:

> [8a] Expressions involving operands of interchange or extended 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 Part 2 of Technical Specification 18661.

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 generic real 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 suggests new **<float.h>** macros, analogous to the macros for generic floating types, that characterize the data-interchange types and the extended floating types. It also suggests macros to indicate which data-interchange types are provided as interchange floating types.

Suggested changes to C11:

In 5.2.4.2.2#7, change the sentence:

All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types.

to:

All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, **FLT_ROUNDS**. **FLT***N*_**IS_ARITH**, and **DEC***N*_**IS_ARITH** have separate names for all floating-point types.

After 5.2.4.2.2#7, add the paragraph

[7a] Some of the macros in **<float.h>** provide characteristics of data-interchange types and extended floating types, as specified in IEC 60559. The prefixes **FLT***N*_ and **DEC***N*_ are used for binary and decimal data-interchange types of width *N*. The prefixes **FLT***N***x**_ and **DEC***N***x**_ are used for binary and decimal extended floating types that extend a basic format of width *N*. For each data-interchange or extended floating type that the implementation provides, **<float.h>** shall define the associated macros. Conversely, for each such type that the implementation does not provide, **<float.h>** shall not define the associated macros.

In 5.2.4.2.2#11, add the following in the bullet defining **FLT_DECIMAL_DIG**, etc.:

FLT*N*_**DECIMAL_DIG**
FLT*N***x**_**DECIMAL_DIG**

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 and data-interchange types with …

Add the following after 5.2.4.2.2#13:

[13a] Whether supported data-interchange types (**_Float***N* and **_Decimal***N*) are further supported as interchange floating types is characterized by the implementation-defined values of **FLT***N*_**IS_ARITH** and **DEC***N*_**IS_ARITH**:

- **0** not supported as a floating type
- **1** supported as a floating type

[13b] In the following lists, 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.

[13c] The integer values given in the following lists shall be expressed by constant expressions suitable for use in **#if** preprocessing directives:

— number of digits 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* binary digits can be rounded to a floating-point number with *n* decimal digits and back again without change to the value, ceiling(1 + *p* $\log_{10}$ 2)

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* binary digits and back again without change to the *q* decimal digits, $\mathrm{floor}((p - 1) \log_{10} 2)$

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, ceiling($\log_{10} 2^{emin-1}$)

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, $\mathrm{floor}(\log_{10}((1 - 2^{-p})2^{\mathrm{emax}}))$

FLT*N*_MAX_10_EXP
FLT*N*X_MAX_10_EXP

[13d] The values given in the following list shall be replaced by constant expressions:

— 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}

FLT*N*_MIN
FLT*N*X_MIN
DEC*N*_MIN
DEC*N*X_MIN

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

FLT*N*_TRUE_MIN
FLT*N*X_TRUE_MIN
DEC*N*_TRUE_MIN
DEC*N*X_TRUE_MIN

8 Conversions

The following suggested change to C11 supports the IEC 60559 restrictions against operands whose types do not have the same radix or such that neither type is a subset of (or equivalent to) the other.

Suggested change to C11:

In 6.3.1.8#1, replace the first 3 items after "This pattern is called the *usual arithmetic conversions*:":

> 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, then the other operand shall not have generic or 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 generic floating type, the other operand is converted, without change of type domain, to a type whose corresponding real type is that same generic 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.

The following suggested change to C11 provides conversions between data-interchange types and other data-interchange types and real floating types.

After 6.3.2.3, add the subclause:

6.3.2.3a Data-interchange types

[1] Any supported data-interchange type can be converted to and from any supported data-interchange type and any real floating type, with rounding to IEC 60559 formats as specified in IEC 60559.

9 Constants

The following suggested changes to C11 provide suffixes that designate constants of data-interchange types and extended floating types.

Suggested changes to C11:

Change *floating-suffix* in 6.4.4.2 from:

floating-suffix: one of
f l F L

to:

floating-suffix: one of
f l F L f*N* **F***N* **f***N***x F***N***x d***N* **D***N* **d***N***x D***N***x**

Add the following paragraph after 6.4.4.2#2:

Constraints

[2a] A *floating-suffix* **d***N*, **D***N*, **d***N***x**, or **D***N***x** shall not be used in a *hexadecimal-floating-constant*.

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

Add the following paragraph after 6.4.4.2#4:

[4a] If a floating constant is suffixed by **f***N* or **F***N*, it has type **_Float***N*. If suffixed by **f***N***x** or **F***N***x**, it has type **_Float***N***x**. If suffixed by **d***N* or **D***N*, it has type **_Decimal***N*. If suffixed by **d***N***x** or **D***N***x**, it has type **_Decimal***N***x**.

Add the following paragraph after 6.4.4.2#5:

[5a] Decimal floating-point constants that have the same numerical value but different quantum exponents have distinguishable internal representations. The quantum exponent is specified to be the same as for the corresponding **strtod***N* or **strtod***N***x** function for the same numeric string.

10 Expressions

The following suggested change to C11 is intended to ensure that data-interchange types can be converted to and from real floating types and data-interchange types, by assignment, cast, argument passing, and function return.

Suggested changes to C11:

At the end of 6.5.16.1#1, append the bullet:

— the left operand has atomic, qualified, or unqualified real floating type or data-interchange type, and the right has real floating type or data-interchange type;

The following suggested changes to C11 specify that certain arithmetic operators need not handle operands of data-interchange type.

Suggested changes to C11:

Change 6.5.3.3#1 from:

[1] The operand of the unary **+** or **–** operator shall have arithmetic type; of the **~** operator, integer type; of the **!** operator, scalar type.

to:

[1] The operand of the unary **+** or **–** operator shall have arithmetic type; of the **~** operator, integer type; of the **!** operator, floating or pointer type.

Change 6.5.13#2 from:

[2] Each of the operands shall have scalar type.

to:

[2] Each of the operands shall have floating or pointer type.

Change 6.5.14#2 from:

[2] Each of the operands shall have scalar type.

to:

[2] Each of the operands shall have floating or pointer type.

Change 6.5.15#2 from:

[2] The first operand shall have scalar type.

to:

[2] The first operand shall have floating or pointer type.

In 7.2.1.1#1, change:

```
void assert(scalar expression);
```

to:

`void assert(`*floating or pointer* **`expression);`**

In 7.2.1.1#2, change the second sentence from:

When it is executed, if **`expression`** (which shall have a scalar type) is false …

to:

When it is executed, if **`expression`** (which shall have a floating or pointer type) is false …

11 Mathematics `<math.h>`

This clause suggests changes to C11 to include functions and macros for interchange and extended floating types. Binary interchange floating types and binary extended floating types are supported by functions and macros corresponding to all those specified for generic floating types (**`float`**, **`double`**, and **`long double`**) in C11 and Part 1 of Technical Specification 18661. Decimal interchange floating types and decimal extended floating types are supported by functions and macros corresponding to all those for the decimal types in Part 2 of Technical Specification 18661. Data-interchange types (including ones that are not interchange floating types) are supported by the classification macros in C11 and Parts 1 and 2 of Technical Specification 18661, and by the **`totalorder`** and **`totalordermag`** functions in Parts 1 and 2.

The list of elementary functions specified in the mathematics library is extended to handle interchange floating types and extended floating types. These include functions specified in C11 (7.12.4, 7.12.5, 7.12.6, 7.12.7, 7.12.8, 7.12.9, 7.12.10, 7.12.11, 7.12.12, and 7.12.13) and in Part 1 of Technical Specification 18661 (14.1, 14.2, 14.3, 14.4, 14.5, 14.8, 14.9, and 14.10). Macros analogous to the **`HUGE_VAL`**, **`INFINITY`**, **`NAN`**, and the **`SNAN`** macros are defined for data-interchange types and extended floating types. Macros **`DEC_INFINITY`** and **`DEC_NAN`** are defined in Part 2 of Technical Specification 18661. With the exception of the floating-point functions listed in 11.2, which have accuracy as specified in IEC 60559, the accuracy of floating-point results is implementation-defined. The implementation may state that the accuracy is unknown. All comparison macros specified in C11 (7.12.14) and in Part 1 of Technical Specification 18661 (14.6) are extended to handle interchange floating types and extended floating types. All classification macros specified in C11 (7.12.3) and in Part 1 of Technical Specification 18661 (14.7) are extended to handle data-interchange types and extended floating types.

Suggested changes to C11:

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.

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 type **float, long double, _Float***N*, **_Float***N***x**, **_Decimal***N*, and **_Decimal***N***x**, respectively.

Add after 7.12#1:

[1a] 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.

Add after 7.12#2:

[2a] For each decimal data-interchange type that the implementation provides, the types

decencodingd*N***_t**
binencodingd*N***_t**

are declared and represents values of the type in the two alternative encodings allowed for decimal formats by the IEC 60559 standard: the encoding (indicated by the prefix **dec**) based on decimal encoding of the significand, or the encoding (indicated by the prefix **bin**) based on binary encoding of the significand. These types are used by the decimal re-encoding functions (7.12.11).

Add at the end of 7.12#3 the following macros:

[3] … For each data-interchange type (**_Float***N* and **_Decimal***N*) that the implementation provides, the corresponding one of the macros

HUGE_VAL_F*N*
HUGE_VAL_D*N*

Is defined and expands to a constant expression of the type representing positive infinity. The macros

HUGE_VAL_F*N***x**
HUGE_VAL_D*N***x**

expand to a constant expressions of types **_Float***N***x** and **_Decimal***N***x**, respectively, representing positive infinity.

After 7.12#5, add the following:

[5b] For each data-interchange type (**_Float***N* and **_Decimal***N*) that the implementation provides, the corresponding one of the signaling NaN macros

SNANF*N*
SNAND*N*

is defined and expands into a constant expression of the type representing a signaling NaN. The signaling NaN macros

SNANF*N***x**
SNAND*N***x**

expand into constant expressions of types **_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.

Add at the end of 7.12 paragraph 7 the following macros.

[7] … The macros

FP_FAST_FMAF*N*
FP_FAST_FMAD*N*

are, respectively, **_Float***N* and **_Decimal***N* analogs of **FP_FAST_FMA**. The macros

FP_FAST_FMAF*N*X
FP_FAST_FMAD*N*X

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

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

7.12.4 Trigonometric functions

```
_Float N acosf N (_Float N x);
_Float Nx acosf Nx(_Float Nx x);
_Decimal N acosd N (_Decimal N x);
_Decimal Nx acosd Nx(_Decimal Nx x);

_Float N asinf N (_Float N x);
_Float Nx asinf Nx(_Float Nx x);
_Decimal N asind N (_Decimal N x);
_Decimal Nx asind Nx(_Decimal Nx x);

_Float N atanf N (_Float N x);
_Float Nx atanf Nx(_Float Nx x);
_Decimal N atand N (_Decimal N x);
_Decimal Nx atand Nx(_Decimal Nx x);

_Float N atan2f N (_Float N y, _Float N x);
_Float Nx atan2f Nx(_Float Nx y, _Float Nx x);
_Decimal N atan2d N (_Decimal N y, _Decimal N x);
_Decimal Nx atan2d Nx(_Decimal Nx y, _Decimal Nx x);

_Float N cosf N (_Float N x);
_Float Nx cosf Nx(_Float Nx x);
_Decimal N cosd N (_Decimal N x);
_Decimal Nx cosd Nx(_Decimal Nx x);

_Float N sinf N (_Float N x);
_Float Nx sinf Nx(_Float Nx x);
_Decimal N sind N (_Decimal N x);
_Decimal Nx sind Nx(_Decimal Nx x);

_Float N tanf N (_Float N x);
_Float Nx tanf Nx(_Float Nx x);
_Decimal N tand N (_Decimal N x);
_Decimal Nx tand Nx(_Decimal Nx x);
```

7.12.5 Hyperbolic functions

```
_Float N acoshf N (_Float N x);
_Float Nx acoshf Nx(_Float Nx x);
_Decimal N acoshd N (_Decimal N x);
_Decimal Nx acoshd Nx(_Decimal Nx x);

_Float N asinhf N (_Float N x);
_Float Nx asinhf Nx(_Float Nx x);
_Decimal N asinhd N (_Decimal N x);
_Decimal Nx asinhd Nx(_Decimal Nx 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);
_DecimalNNx 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 frexpfN (_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);

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

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

```
_Decimal Nx ldexpd Nx (_Decimal Nx value, int exp);

_Float N logf N (_Float N x);
_Float Nx logf Nx(_Float Nx x);
_Decimal N logd N (_Decimal N x);
_Decimal Nx logd Nx(_Decimal Nx x);

_Float N log10f N (_Float N x);
_Float Nx log10f Nx(_Float Nx x);
_Decimal N log10d N (_Decimal N x);
_Decimal Nx log10d Nx(_Decimal Nx x);

_Float N log1pf N (_Float N x);
_Float Nx log1pf Nx(_Float Nx x);
_Decimal N log1pd N (_Decimal N x);
_Decimal Nx log1pd Nx(_Decimal Nx x);

_Float N log2f N (_Float N x);
_Float Nx log2f Nx(_Float Nx x);
_Decimal N log2d N (_Decimal N x);
_Decimal Nx log2d Nx(_Decimal Nx x);

_Float N logbf N (_Float N x);
_Float Nx logbf Nx(_Float Nx x);
_Decimal N logbd N (_Decimal N x);
_Decimal Nx logbd Nx(_Decimal Nx x);

_Float N modff N (_Float N x, _Float N *iptr);
_Float Nx modff Nx(_Float Nx x, _Float Nx *iptr);
_Decimal N modfd N (_Decimal N x, _Decimal N *iptr);
_Decimal Nx modfd Nx(_Decimal Nx x, _Decimal Nx *iptr);

_Float N scalbnf N (_Float N value, int exp);
_Float Nx scalbnf N (_Float Nx value, int exp);
_Decimal N scalbnd N (_Decimal N value, int exp);
_Decimal Nx scalbnd Nx (_Decimal Nx value, int exp);

_Float N scalblnf N (_Float N value, long int exp);
_Float Nx scalblnf N (_Float Nx value, long int exp);
_Decimal N scalblnd N (_Decimal N value, long int exp);
_Decimal Nx scalblnd Nx (_Decimal Nx value, long int exp);
```

7.12.7 Power and absolute-value functions

```
_Float N cbrtf N (_Float N x);
_Float Nx cbrtf Nx(_Float Nx x);
_Decimal N cbrtd N (_Decimal N x);
_Decimal Nx cbrtd Nx(_Decimal Nx x);

_Float N fabsf N (_Float N x);
_Float Nx fabsf Nx(_Float Nx x);
_Decimal N fabsd N (_Decimal N x);
_Decimal Nx fabsd Nx(_Decimal Nx x);

_Float N hypotf N (_Float N x, _Float N y);
_Float Nx hypotf Nx(_Float Nx x, _Float Nx y);
```

```
_Decimal N hypotd N (_Decimal N x, _Decimal N y);
_Decimal Nx hypotd Nx(_Decimal Nx x, _Decimal Nx y);

_Float N powf N (_Float N x, _Float N y);
_Float Nx powf Nx(_Float Nx x, _Float Nx y);
_Decimal N powd N (_Decimal N x, _Decimal N y);
_Decimal Nx powd Nx(_Decimal Nx x, _Decimal Nx y);


_Float N sqrtf N (_Float N x);
_Float Nx sqrtf Nx(_Float Nx x);
_Decimal N sqrtd N (_Decimal N x);
_Decimal Nx sqrtd Nx(_Decimal Nx x);
```

7.12.8 Error and gamma functions

```
_Float N erff N (_Float N x);
_Float Nx erff Nx(_Float Nx x);
_Decimal N erfd N (_Decimal N x);
_Decimal Nx erfd Nx(_Decimal Nx x);

_Float N erfcf N (_Float N x);
_Float Nx erfcf Nx(_Float Nx x);
_Decimal N erfcd N (_Decimal N x);
_Decimal Nx erfcd Nx(_Decimal Nx x);

_Float N lgammaf N (_Float N x);
_Float Nx lgammaf Nx(_Float Nx x);
_Decimal N lgammad N (_Decimal N x);
_Decimal Nx lgammad Nx(_Decimal Nx x);

_Float N tgammaf N (_Float N x);
_Float Nx tgammaf Nx(_Float Nx x);
_Decimal N tgammad N (_Decimal N x);
_Decimal Nx tgammad Nx(_Decimal Nx x);
```

7.12.9 Nearest integer functions

```
_Float N ceilf N (_Float N x);
_Float Nx ceilf Nx(_Float Nx x);
_Decimal N ceild N (_Decimal N x);
_Decimal Nx ceild Nx(_Decimal Nx x);

_Float N floorf N (_Float N x);
_Float Nx floorf Nx(_Float Nx x);
_Decimal N floord N (_Decimal N x);
_Decimal Nx floord Nx(_Decimal Nx x);

_Float N nearbyintf N (_Float N x);
_Float Nx nearbyintf Nx(_Float Nx x);
_Decimal N nearbyintd N (_Decimal N x);
_Decimal Nx nearbyintd Nx(_Decimal Nx x);

_Float N rintf N (_Float N x);
_Float Nx rintf Nx(_Float Nx x);
_Decimal N rintd N (_Decimal N x);
_Decimal Nx rintd Nx(_Decimal Nx x);
```

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

long long int llrintfN (_FloatN x);
long long int llrintfNx (_FloatN x);
long long int llrintdN (_DecimalN x);
long long int llrintdNx (_DecimalN 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 (_FloatN x);
long int lrounddN (_DecimalN x);
long int lrounddNx (_DecimalN x);

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

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

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

_FloatN canonicalizefN (_FloatN x);
_FloatNx canonicalizefNx(_FloatNx x);
_DecimalN canonicalizedN (_DecimalN x);
_DecimalNx canonicalizedNx(_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);

int quantexpdN (_DecimalN x);
int quantexpdNx(_DecimalNx x);

decencodingdN_t encodedecdN (_DecimalN x);
_DecimalN decodedecdN (decencodingdN_t x);
binencodingdN_t encodebindN (_DecimalN x);
```

```
_Decimal N decodebind N (binencodingd N _t x);
```

7.12.12 Maximum, minimum, and positive difference functions

```
_Float N fdimf N (_Float N x, _Float N y);
_Float Nx fdimf Nx(_Float Nx x, _Float Nx y);
_Decimal N fdimd N (_Decimal N x, _Decimal N y);
_Decimal Nx fdimd Nx(_Decimal Nx x, _Decimal Nx y);

_Float N fmaxf N (_Float N x, _Float N y);
_Float Nx fmaxf Nx(_Float Nx x, _Float Nx y);
_Decimal N fmaxd N (_Decimal N x, _Decimal N y);
_Decimal Nx fmaxd Nx(_Decimal Nx x, _Decimal Nx y);

_Float N fminf N (_Float N x, _Float N y);
_Float Nx fminf Nx(_Float Nx x, _Float Nx y);
_Decimal N fmind N (_Decimal N x, _Decimal N y);
_Decimal Nx fmind Nx(_Decimal Nx x, _Decimal Nx y);

_Float N fmaxmagf N (_Float N x, _Float N y);
_Float Nx fmaxmagf Nx(_Float Nx x, _Float Nx y);
_Decimal N fmaxmagd N (_Decimal N x, _Decimal N y);
_Decimal Nx fmaxmagd Nx(_Decimal Nx x, _Decimal Nx y);

_Float N fminmagf N (_Float N x, _Float N y);
_Float Nx fminmagf Nx(_Float Nx x, _Float Nx y);
_Decimal N fminmagd N (_Decimal N x, _Decimal N y);
_Decimal Nx fminmagd Nx(_Decimal Nx x, _Decimal Nx y);
```

7.12.13 Floating multiply-add

```
_Float N fmaf N (_Float N x, _Float N y, _Float N z);
_Float N fmaf Nx (_Float Nx x, _Float Nx y, _Float Nx z);
_Decimal N fmad N (_Decimal N x, _Decimal N y, _Decimal N z);
_Decimal Nx fmad Nx (_Decimal Nx x, _Decimal Nx y, _Decimal Nx z);
```

7.12.14 Functions that round result to narrower format

```
_Float M f M addf N (_Float N x, _Float N y);   // M < N
_Float M f M addf Nx (_Float Nx x, _Float Nx y);   // M <= N
_Float Mx f Mx addf N (_Float N x, _Float N y); // M < N
_Float Mx f Mx addf Nx (_Float Nx x, _Float Nx y); // M < N
_Decimal M d M add d N (_Decimal N x, _Decimal N y);// M < N
_Decimal M d M add d Nx (_Decimal Nx x, _Decimal Nx y); // M <= N
_Decimal Mx d Mx add d N (_Decimal N x, _Decimal N y);  // M < N
_Decimal Mx d Mx add d Nx (_Decimal Nx x, _Decimal Nx y);  // M < N

_Float M f M subf N (_Float N x, _Float N y);   // M < N
_Float M f M subf Nx (_Float Nx x, _Float Nx y);   // M <= N
_Float Mx f Mx subf N (_Float N x, _Float N y); // M < N
_Float Mx f Mx subf Nx (_Float Nx x, _Float Nx y); // M < N
_Decimal M d M subd N (_Decimal N x, _Decimal N y);// M < N
_Decimal M d M subd Nx (_Decimal Nx x, _Decimal Nx y); // M <= N
_Decimal Mx d Mx subd N (_Decimal N x, _Decimal N y);  // M < N
_Decimal Mx d Mx subd Nx (_Decimal Nx x, _Decimal Nx 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 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 dMxdivdN (_DecimalN x);  // M < N
_DecimalMx dMxdivdNx (_DecimalNx x);   // 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 fMxdivfNx (_FloatNx x, _FloatNx y, _FloatNx z);   // M < N
_DecimalM dMfmadN (_DecimalN x, _DecimalN y, _DecimalN z); // M < N
_DecimalM dMdfmadNx (_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
```

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 F.10.12.2 (see Part 1 of Technical Specification 18661), append to paragraph 2:

The **totalorderf***N* and t**otalordermagf***N* functions are declared for each corresponding data-interchange type that the implementation provides.

12 Numeric conversion functions <stdlib.h>

This clause specifies functions to convert between character sequences and the data-interchange types 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 new functions that perform conversions like **snprintf_s** in **<stdio.h>**.

Suggested changes to C11:

After 7.22.1.4, add:

7.22.1.5 The strtof*N*, strtof*N*x, strtod*N*, and strtod*N*x functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_18661_EXT3__
#include <stdlib.h>
_Float*N* strtof*N* (const char * restrict nptr, char ** restrict
   endptr);
_Float*N*x strtof*N*x (const char * restrict nptr, char ** restrict
   endptr);
_Decimal*N* strtod*N* (const char * restrict nptr, char ** restrict
   endptr);
_Decimal*N*x strtod*N*x (const char * restrict nptr, char ** restrict
   endptr);
```

Description

The **strtof***N* and **strtof***N***x** functions are similar to the **strtod** function, except they convert to the types **_Float***N* and **_Float***N***x** respectively. The **strtod***N* and **strtod***N***x** functions are similar to the **strtod64** function, specified in Part 2 of Technical Specification 18661, except they convert to the types **_Decimal***N* and **_Decimal***N***x** respectively.

Returns

The **strtof***N* and **strtof***N***x** functions return values similar to the **strtod** function, except in the types **_Float***N* and **_Float***N***x** respectively. The **strtod***N* and **strtod***N***x** functions return values similar to the **strtod64** function, except in the types **_Decimal***N* and **_Decimal***N***x** respectively

7.22.1.6 The `strfromf`*N*, `strfromf`*N*`x`, `strfromd`*N*, and `strfromd`*N*`x` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_18661_EXT3__
#include <stdlib.h>
int strfromfN (char * restrict s, rsize_t n, const char * restrict
   format, _FloatN fp);
int strfromfNx(char * restrict s, rsize_t n, const char * restrict
   format, _FloatNx fp);
int strfromdN (char * restrict s, rsize_t n, const char * restrict
   format, _DecimalN fp);
int strfromdNx(char * restrict s, rsize_t n, const char * restrict
   format, _DecimalNx fp);
```

Description

These functions are equivalent to **`snprintf_s(s, n, format, fp)`** (K.3.5.3.5), except the type is implied by the function suffix and the **`format`** string contains no length modifier.

Returns

These functions return the value that would be returned by **`snprintf_s(s, n, format, fp)`**.

13 Complex arithmetic `<complex.h>`

This clause specifies complex functions for corresponding real types that are interchange and extended floating types.

Suggested changes to C11:

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

```
_FloatN complex cpowfN (_FloatN complex z, _FloatN complex y);
_FloatNx complex cpowfNx(_FloatNx complex z, _FloatNx complex y);

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

7.3.9 Manipulation functions

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

14 Type-generic macros <tgmath.h>

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

Suggested changes to C11:

In 7.25, replace paragraphs 2 and 3:

> [2] Of the **<math.h>** and **<complex.h>** functions without an **f** (**float**) or **l** (**long double**) suffix, several have one or more parameters whose corresponding real type is **double.** For each such function, except **modf,** there is a corresponding *type-generic macro*.313) The parameters whose corresponding real type is **double** in the function synopsis are *generic parameters.* Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters.314)
>
> [3] Use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:
>
> — First, if any argument for generic parameters has type **long double,** the type determined is **long double.**
>
> — Otherwise, if any argument for generic parameters has type **double** or is of integer type, the type determined is **double.**
>
> — Otherwise, the type determined is **float.**

with:

> [2] This clause specifies a many-to-one correspondence of functions in **<math.h>** and **<complex.h>** with a *type-generic macro*.313) Use of the type-generic macro invokes a

corresponding function whose type is determined by the types of the arguments for particular parameters called the *generic parameters*.314)

[3] Of the **<math.h>** and **<complex.h>** functions without a type suffix, several have one or more parameters whose corresponding real type is **double.** For each such function, except **modf**, there is a corresponding type-generic macro.*313)* The parameters whose corresponding real type is **double** in the function synopsis are generic parameters.

[3a] Some of the **<math.h>** functions for decimal floating types have no unsuffixed counterpart. Of these functions with a **d64** suffix, some have one or more parameters whose type is **_Decimal64**. For each such function, except **encodedecd64** and **encodebind64**, there is a corresponding type-generic macro. The parameters whose real type is **_Decimal64** in the function synopsis are generic parameters.

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

[3c] Use of a type-generic macro invokes a function whose generic parameters have the corresponding real type determined by the corresponding real types of the arguments as follows:

— 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* ≥ 64 or **_Decimal***N***x** where *N* ≥ 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* ≥ 128, or **_Float***N***x** where *N* ≥ 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**.

If neither **<math.h>** nor **<complex.h>** define a function whose generic parameters have the determined corresponding real type, the behavior is undefined.

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#5, replace the last sentence:

If all arguments for generic parameters are real, then use of the macro invokes a real function; otherwise, use of the macro results in undefined behavior.

with:

If all arguments for generic parameters are real, then use of the macro invokes a real function (provided **<math.h>** defines a function of the determined type); otherwise, use of the macro results in undefined behavior.

In 7.25#6, replace the last sentence:

Use of the macro with any real or complex argument invokes a complex function.

with:

Use of the macro with any argument of generic floating type, binary floating type, 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#6, add:

[6a] For each **d64**-suffixed function in **<math.h>** (except **encodedecd64** and **encodebind64**) that does not have an unsuffixed counterpart, the corresponding type-generic macro has the name of the function, but without the suffix. These type-generic macros are:

<math.h> function	type-generic macro
quantized64	**quantize**
samequantumd64	**samequantum**
quantexpd64	**quantexp**

Use of the macro with a generic floating or complex argument or with only integer type arguments results in undefined behavior.

[6b] For an implementation that supports Parts 1 and 3 (but not Part 2) of Technical Specification 18661 and that provides the following types:

type	IEC 60559 format
float	binary32
double	binary64
long double	binary128
_Float32	binary32
_Float64	binary64

`_Float128`	binary128
`_Float32x`	binary64
`_Float64x`	**b**inary128

a type-generic macro **`cbrt`** that conforms to the specification in this clause and that is affected by constant rounding modes as specified in Part 1 of Technical Specification 18661 could be implemented as follows:

```
#if defined(__STDC_WANT_IEC_18661_EXT3)
   #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_18661_EXT3__
```

In 7.25#7, append to the declarations:

```
#if __STDC_IEC_60559_TYPES__ >= 201ymmL
_Float32x f32x;
_Float64 f64;
_Float128 f128;
_Float64 complex f64c;
#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 (the sets are not equivalent)
`pow(f32x, n)`	`powf32x(f32x, n)`

Bibliography

[1] ISO/IEC 9899:2011, *Information technology — Programming languages, their environments and system software interfaces — Programming Language C*

[2] ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*

[3] 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*

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

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

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

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