ISO/IEC JTC 1/SC 22/WG 14 N1809

Date: yyyy-mm-dd

Reference number of document: TECHNICAL SPECIFICATION

ISO/IEC TS 18661-1

WG 14 N2315Committee identification: ISO/IEC JTC 1/SC 22/WG 14

Secretariat: ANSI

CFP Working DraftSecond edition
For C2X integration
201y-mm-dd2018-11-12

Information Technologytechnology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 1: Binary floating-point arithmetic

Part 1:
Binary floating-point arithmetic

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

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*Partie 1:*

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

Binaire arithmétique flottante

Reference number
ISO/IEC TS 18661-1:2014(E)

COPYRIGHT PROTECTED DOCUMENT

~~Contents~~

~~Page~~

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing 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 is normally carried out through ISO technical committees. Each member body interested in a subject for which a established by the respective organization to deal with particular fields of technical committee has been established has the right to be represented on that committee. Internationalactivity. 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. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardizationIn 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/directivesInternational 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 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/IEC TS 18661 was prepared by 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 informationCommittee

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

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

— *Part 1: Binary floating-point arithmetic*

— *Part 2: Decimal floating-point arithmetic*

— *Part 3: Interchange and extended types*

— *Part 4: Supplementary functions*

— *Part 5: Supplementary attributes*

~~Part~~ ISO/IEC TS 18661-1 updates ISO/IEC 9899:2011 ~~(~~, *Information technology — Programming ~~languages, their environments and system software interfaces — Programming~~ Language C*~~), Annex~~, annex F in particular, to support all required features of ISO/IEC/IEEE 60559:2011 ~~(~~, *Information technology — Microprocessor Systems — Floating-point arithmetic*~~).~~.

~~Part~~ ISO/IEC TS 18661-2 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~~ISO/IEC TS 18661-3, ISO/IEC TS 18661-4, and ISO/IEC TS 18661-5 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 the following:

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

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

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

4 Provide direct support for

a. Execution-time diagnosis of anomalies

b. Smoother handling of exceptions

c. Interval arithmetic at a reasonable cost

5 Provide for development of

a. Standard elementary functions such as exp and cos

b. Very high precision (multiword) arithmetic

c. Coupling of numerical and symbolic algebraic computation

6 Enable rather than preclude further refinements and extensions.

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

— *formats* – for binary floating-point data, including representations for Not-a-Number (NaN) and signed infinities and zeros

— *operations* – basic arithmetic operations (addition, multiplication, etc.) on the format data to compose a well-defined, closed arithmetic system; also specified conversions between floating-point formats and decimal character sequences, and a few auxiliary operations

— *context* – status flags for detecting exceptional conditions (invalid operation, division by zero, overflow, underflow, and inexact) and controls for choosing different rounding methods

The ISO/IEC/IEEE 60559:2011 international standard is equivalent to the IEEE 754-2008 standard for floating-point arithmetic, which is a major revision to IEEE 754-1985.

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

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

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

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

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

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

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

C support for IEC 60559

The C standard specifies floating-point arithmetic using an abstract model. The representation of a floating-point number is specified in an abstract form where the constituent components (sign, exponent, significand) of the representation are defined but not the internals of these components. In particular, the exponent range, significand size, and the base (or radix) are implementation-defined. This allows flexibility for an implementation to take advantage of its underlying hardware architecture. Furthermore, certain behaviors of operations are also implementation-defined, for example in the area of handling of special numbers and in exceptions.

The reason for this approach is historical. At the time when C was first standardized, before the floating-point standard was established, there were various hardware implementations of floating-point arithmetic in common use. Specifying the exact details of a representation would have made most of the existing implementations at the time not conforming.

Beginning with ISO/IEC 9899:1999 (C99), C has included an optional second level of specification for implementations supporting the floating-point standard. C99, in conditionally normative ~~Annex~~annex F, introduced nearly complete support for the IEC 60559:1989 standard for binary floating-point arithmetic. Also, C99's informative ~~Annex~~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 it is still based on IEC 60559:1989. C11 ~~upgrades Annex~~upgraded annex G from "informative" to "conditionally normative".

ISO/IEC ~~Technical Report~~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 ~~this Technical Specification~~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~~parts.

~~Part~~ ISO/IEC TS 18661-1~~, this document,~~ 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~~annex F as enhanced by the changes in ~~Part~~ ISO/IEC TS 18661-1.

~~Part~~ ISO/IEC TS 18661-2 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~~ ISO/IEC TS 18661-2.

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

Information ~~Technology~~technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — ~~Part 1: Binary floating-point arithmetic~~

Part 1: Binary floating-point arithmetic

1 Scope

This ~~document, Part 1~~part of ISO/IEC ~~Technical Specification~~TS 18661~~,~~ extends programming language C to support binary floating-point arithmetic conforming to ISO/IEC/IEEE 60559:2011. It covers all requirements of IEC 60559 as they pertain to C floating types that use IEC 60559 binary formats.

This document does not cover decimal floating-point arithmetic, nor does it cover most optional features of IEC 60559.

This document is primarily an update to IEC 9899:2011 (C11), normative Annex F (IEC 60559 floating-point arithmetic). However, it proposes that the new interfaces that are suitable for general implementations be added in the Library clauses of C11. Also it includes a few auxiliary changes in C11 where the specification is problematic for IEC 60559 support.

2 Conformance

An implementation conforms to ~~Part 1~~this part of ~~Technical Specification~~ISO/IEC TS 18661 if

a) It meets the requirements for a conforming implementation of C11 with all the changes to C11 specified in Part 1 of Technical Specification 18661; and

b) It defines `__STDC_IEC_60559_BFP__` to `20y~~1~~ymmL`.

3 Normative references

The following ~~referenced~~ documents, in whole or in part, are normatively referenced in this document and are indispensable for ~~the~~its application~~ of this document. Only the editions~~. For dated references, only the edition cited ~~apply~~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)~~

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

5 C standard conformance

5.1 Freestanding implementations

The following change to C11 expands the conformance requirements for freestanding implementations so that they might conform to this ~~Part~~part of Technical Specification18661.

Change to C11:

Insert after the third sentence of 4#6:

> The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines **`__STDC_IEC_60559_BFP__`** may also use features in the contents of the standard headers **`<fenv.h>`** and **`<math.h>`** and the numeric conversion functions (7.22.1) of the standard header **`<stdlib.h>`**. All identifiers that are reserved when **`<stdlib.h>`** is included in a hosted implementation are reserved when it is included in a freestanding implementation.

5.2 Predefined macros

The following changes to C11 obsolesce **`__STDC_IEC_559__`**, the current conformance macro for Annex F, in favor of **`__STDC_IEC_60559_BFP__`**, for consistency with other conformance macros and to distinguish its application to binary floating-point arithmetic. The macro **`__STDC_IEC_559__`** is retained as obsolescent, for compatibility with existing programs.

Changes to C11:

In 6.10.8.3#1, before:

> **`__STDC_IEC_559__`** The integer constant **`1`**, intended to indicate conformance to Annex F (IEC 60559 binary floating-point arithmetic).

insert:

> **__STDC_IEC_60559_BFP__** The integer constant **20y~~1~~ymmL**, intended to indicate conformance to Annex F (IEC 60559 binary floating-point arithmetic).

In 6.10.8.3#1, append to the **__STDC_IEC_559__** item:

> Use of this macro is an obsolescent feature.

The following changes to C11 obsolesce **__STDC_IEC_559_COMPLEX__**, the current conformance macro for Annex G, in favor of **__STDC_IEC_60559_COMPLEX__**, for consistency with other conformance macros.

Changes to C11:

In 6.10.8.3#1, ~~after~~ before the **__STDC_IEC_559_COMPLEX__** ~~**__STDC_IEC_559__**~~ item, insert the item:

> **__STDC_IEC_60559_COMPLEX__** The integer constant **20y~~1~~ymmL**, intended to indicate conformance to the specifications in annex G (IEC 60559 compatible complex arithmetic).

In 6.10.8.3#1, append to the **__STDC_IEC_559_COMPLEX__** item:

> Use of this macro is an obsolescent feature.

5.3 Standard headers

The new identifiers added to C11 library headers by this Part of Technical Specification 18661 are defined or declared by their respective headers only if **__STDC_WANT_IEC_60559_BFP_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 list these identifiers in each applicable library subclause.

Changes to C11:

After 5.2.4.2.1#1, insert the paragraph:

> [1a] The following identifiers are defined only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<limits.h>** is first included:

```
CHAR_WIDTH      USHRT_WIDTH     ULONG_WIDTH
SCHAR_WIDTH     INT_WIDTH       LLONG_WIDTH
UCHAR_WIDTH     UINT_WIDTH      ULLONG_WIDTH
SHRT_WIDTH      LONG_WIDTH
```

After 5.2.4.2.2#6, insert the paragraph:

> [6a] The following identifier is defined only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<float.h>** is first included:

```
CR_DECIMAL_DIG
```


After 7.6#3, insert the paragraph:

[3a] The following identifiers are defined or declared only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<fenv.h>** is first included:

```
femode_t                    fetestexceptflag
FE_DFL_MODE                 fegetmode
FE_SNANS_ALWAYS_SIGNAL      fesetmode
fesetexcept
```

After 7.12#1, insert the paragraph:

[1a] The following identifiers are defined or declared only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<math.h>** is first included:

```
FP_INT_UPWARD                FP_FAST_FSUBL
FP_INT_DOWNWARD              FP_FAST_DSUBL
FP_INT_TOWARDZERO            FP_FAST_FMUL
FP_INT_TONEARESTFROMZERO     FP_FAST_FMULL
FP_INT_TONEAREST             FP_FAST_DMULL
FP_LLOGB0                    FP_FAST_FDIV
FP_LLOGBNAN                  FP_FAST_FDIVL
SNANF                        FP_FAST_DDIVL
SNAN                         FP_FAST_FFMA
SNANL                        FP_FAST_FFMAL
FP_FAST_FADD                 FP_FAST_DFMAL
FP_FAST_FADDL                FP_FAST_FSQRT
FP_FAST_DADDL                FP_FAST_FSQRTL
FP_FAST_FSUB                 FP_FAST_DSQRTL
```

iseqsig	**fmaxmagf**	**ffmal**
iscanonical	**fmaxmagl**	**dfmal**
issignaling	**fminmag**	**fsqrt**
issubnormal	**fminmagf**	**fsqrtl**
iszero	**fminmagl**	**dsqrtl**
fromfp	**nextup**	**totalorder**
fromfpf	**nextupf**	**totalorderf**
fromfpl	**nextupl**	**totalorderl**
ufromfp	**nextdown**	**totalordermag**
ufromfpf	**nextdownf**	**totalordermagf**
ufromfpl	**nextdownl**	**totalordermagl**
fromfpx	**fadd**	**canonicalize**
fromfpxf	**faddl**	**canonicalizef**
fromfpxl	**daddl**	**canonicalizel**
ufromfpx	**fsub**	**getpayload**
ufromfpxf	**fsubl**	**getpayloadf**
ufromfpxl	**dsubl**	**getpayloadl**
roundeven	**fmul**	**setpayload**
roundevenf	**fmull**	**setpayloadf**
roundevenl	**dmull**	**setpayloadl**
llogb	**fdiv**	**setpayloadsig**
llogbf	**fdivl**	**setpayloadsigf**
llogbl	**ddivl**	**setpayloadsigl**
fmaxmag	**ffma**	

After 7.20#4, insert the paragraph:

[4a] The following identifiers are defined only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<stdint.h>** is first included:

INT*N*_WIDTH	**UINT_FAST*N*_WIDTH**	**PTRDIFF_WIDTH**
UINT*N*_WIDTH	**INTPTR_WIDTH**	**SIG_ATOMIC_WIDTH**
INT_LEAST*N*_WIDTH	**UINTPTR_WIDTH**	**SIZE_WIDTH**
UINT_LEAST*N*_WIDTH	**INTMAX_WIDTH**	**WCHAR_WIDTH**
INT_FAST*N*_WIDTH	**UINTMAX_WIDTH**	**WINT_WIDTH**

After 7.22#1, insert the paragraph:

[1a] The following identifiers are declared only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<stdlib.h>** is first included:

strfromd	**strfromf**	**strfroml**

After 7.25#1, insert the paragraph:

[1a] The following identifiers are defined as type-generic macros only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<tgmath.h>** is first included:

`roundeven`	`fromfpx`	`fmul`
`llogb`	`ufromfpx`	`dmul`
`fmaxmag`	`totalorder`	`fdiv`
`fminmag`	`totalordermag`	`ddiv`
`nextup`	`fadd`	`ffma`
`nextdown`	`dadd`	`dfma`
`fromfp`	`fsub`	`fsqrt`
`ufromfp`	`dsub`	`dsqrt`

After 7.1.2#4, insert:

[4a] Some standard headers define or declare identifiers contingent on whether certain macros whose names begin with `__STDC_WANT_IEC_60559_` and end with `_EXT__` are defined (by the user) at the point in the code where the header is first included. Within a preprocessing translation unit, the same set of such macros shall be defined for the first inclusion of all such headers.

6 Revised floating-point standard

C11 Annex F specifies C language support for the floating-point arithmetic of IEC 60559:1989. This document proposes changes to C11 to bring Annex F into alignment with IEC 60559:2011. The changes to C11 below update the introduction to Annex F to acknowledge the revision to IEC 60559.

Changes to C11:

Change F.1 from:

F.1 Introduction

[1] This annex specifies C language support for the IEC 60559 floating-point standard. The *IEC 60559 floating-point standard* is specifically *Binary floating-point arithmetic for microprocessor systems, second edition* (IEC 60559:1989), previously designated IEC 559:1989 and as *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE 754−1985). *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854−1987) generalizes the binary standard to remove dependencies on radix and word length. *IEC 60559* generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc. An implementation that defines `__STDC_IEC_559__` shall conform to the specifications in this annex.356) Where a binding between the C language and IEC60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise. Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values.

to:

F.1 Introduction

[1] This annex specifies C language support for the IEC 60559 floating-point standard. The *IEC 60559 floating-point standard* is specifically *Floating-point arithmetic* (ISO/IEC/IEEE 60559:2011), also designated as *IEEE Standard for Floating-Point Arithmetic* (IEEE 754−2008). The IEC 60559 floating-point standard supersedes the IEC 60559:1989 binary arithmetic standard, also designated as *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754−1985). *IEC 60559* generally refers to the floating-point standard, as in IEC 60559 operation, IEC 60559 format, etc.

[2] The IEC 60559 floating-point standard specifies decimal, as well as binary, floating-point arithmetic. It supersedes *IEEE Standard for Radix-Independent Floating-Point Arithmetic* (ANSI/IEEE 854−1987), which generalized the binary arithmetic standard (IEEE 754-1985) to remove dependencies on radix and word length.

[3] An implementation that defines **`__STDC_IEC_60559_BFP__`** to **`20y~~1~~ymmL`** shall conform to the specifications in this annex.356) Where a binding between the C language and IEC 60559 is indicated, the IEC 60559-specified behavior is adopted by reference, unless stated otherwise.

In footnote 356), change "**`__STDC_IEC_559__`**" to "**`__STDC_IEC_60559_BFP__`**".

Note that the last sentence of F.1 which is removed above is inserted into a more appropriate place by a later change (see 12 below).

7 Types

7.1 Terminology

IEC 60559 now includes a 128-bit binary format as one of its three binary basic formats: *binary32*, *binary64*, and *binary128*. The binary128 format continues to meet the less specific requirements for a binary64-extended format, as in the previous IEC 60559. The changes to C11 below reflect the new terminology in IEC 60559; these changes are not substantive.

Changes to C11:

In F.2#1, change the three bullets from:

— The **`float`** type matches the IEC 60559 single format.

— The **`double`** type matches the IEC 60559 double format,

— The **`long double`** type matches an IEC 60559 extended format,357) else a non-IEC 60559 extended format, else the IEC 60559 **`double`** format.

to:

— The **`float`** type matches the IEC 60559 binary32 format.

— The **`double`** type matches the IEC 60559 binary64 format.

— The **`long double`** type matches the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format,357) else a non-IEC 60559 extended format, else the IEC 60559 binary64 format.

In F.2#1, change the sentence after the bullet from:

Any non-IEC 60559 extended format used for the **`long double`** type shall have more precision than IEC 60559 double and at least the range of IEC 60559 double.358)

to:

Any non-IEC 60559 extended format used for the **`long double`** type shall have more precision than IEC 60559 binary64 and at least the range of IEC 60559 binary64.358)

Change footnote 357) from:

> 357) "Extended" is IEC 60559's double-extended data format. Extended refers to both the common 80-bit and quadruple 128-bit IEC 60559 formats.

to:

> 357) IEC 60559 binary64-extended formats include the common 80-bit IEC 60559 format.

In F.2, change the recommended practice from:

> **Recommended practice**
>
> [2] The **`long double`** type should match an IEC 60559 extended format.

to:

> **Recommended practice**
>
> [2] The **`long double`** type should match the IEC 60559 binary128 format, else an IEC 60559 binary64-extended format.

Change footnote 361) from:

> 361) If the minimum-width IEC 60559 extended format (64 bits of precision) is supported, **`DECIMAL_DIG`** shall be at least 21. If IEC 60559 double (53 bits of precision) is the widest IEC 60559 format supported, then **`DECIMAL_DIG`** shall be at least 17. (By contrast, **`LDBL_DIG`** and **`DBL_DIG`** are 18 and 15, respectively, for these formats.)

to:

> 361) If the minimum-width IEC 60559 binary64-extended format (64 bits of precision) is supported, **`DECIMAL_DIG`** shall be at least 21. If IEC 60559 binary64 (53 bits of precision) is the widest IEC 60559 format supported, then **`DECIMAL_DIG`** shall be at least 17. (By contrast, **`LDBL_DIG`** and **`DBL_DIG`** are 18 and 15, respectively, for these formats.)

7.2 Canonical representation

IEC 60559 refers to preferred encodings in a format – or, in C terminology, preferred representations of a type – as *canonical*. Some types also contain redundant or ill-specified representations, which are *non-canonical*. All representations of types with IEC 60559 binary interchange formats are canonical; however, types with IEC 60559 extended formats may have non-canonical encodings. (Types with IEC 60559 decimal interchange formats, covered in Part 2 of Technical Specification 18661, contain non-canonical redundant representations.)

Changes to C11:

In 5.2.4.2.2#3, change the sentence:

> A *NaN* is an encoding signifying Not-a-Number.

to:

> A *NaN* is a value signifying Not-a-Number.

In 5.2.4.2.2 footnote 22, change:

> … the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

to:

> … the terms quiet NaN and signaling NaN are intended to apply to values with similar behavior.

After 5.2.4.2.2#5, add:

> [5a] An implementation may prefer particular representations of values that have multiple representations in a floating type, 6.2.6.1 not withstanding. The preferred representations of a floating type, including unique representations of values in the type, are called *canonical*. A floating type may also contain *non-canonical* representations, for example, redundant representations of some or all of its values, or representations that are extraneous to the floating-point model. Typically, floating-point operations deliver results with canonical representations. IEC 60559 operations deliver results with canonical representations, unless specified otherwise.

In 5.2.4.2.2#5a, attach a footnote to the wording:

> An implementation may prefer particular representations of values that have multiple representations in a floating type, 6.2.6.1 not withstanding.

where the footnote is:

> *) The library operations **`iscanonical`** and **`canonicalize`** distinguish canonical (preferred) representations, but this distinction alone does not imply that canonical and non-canonical representations are of different values.

In 5.2.4.2.2#5a, attach a footnote to the wording:

> A floating type may also contain *non-canonical* representations, for example, redundant representations of some or all of its values, or representations that are extraneous to the floating-point model.

where the footnote is:

> *) Some of the values in the IEC 60559 decimal formats have non-canonical representations (as well as a canonical representation).

8 Operation binding

IEC 60559 includes several new required operations. The change to C11 below shows the complete mapping of IEC 60559 operations to C operators, functions, and function-like macros. The new IEC 60559 operations map to C functions and function-like macros; no new C operators are proposed.

Change to C11:

Replace F.3:

F.3 Operators and functions

[1] C operators and functions provide IEC 60559 required and recommended facilities as listed below.

— The **+**, **−**, *****, and **/** operators provide the IEC 60559 add, subtract, multiply, and divide operations.

— The **sqrt** functions in **<math.h>** provide the IEC 60559 square root operation.

— The **remainder** functions in **<math.h>** provide the IEC 60559 remainder operation. The **remquo** functions in **<math.h>** provide the same operation but with additional information.

— The **rint** functions in **<math.h>** provide the IEC 60559 operation that rounds a floating-point number to an integer value (in the same precision). The **nearbyint** functions in **<math.h>** provide the nearbyinteger function recommended in the Appendix to ANSI/IEEE 854.

— The conversions for floating types provide the IEC 60559 conversions between floating-point precisions.

— The conversions from integer to floating types provide the IEC 60559 conversions from integer to floating point.

— The conversions from floating to integer types provide IEC 60559-like conversions but always round toward zero.

— The **lrint** and **llrint** functions in **<math.h>** provide the IEC 60559 conversions, which honor the directed rounding mode, from floating point to the **long int** and **long long int** integer formats. The **lrint** and **llrint** functions can be used to implement IEC 60559 conversions from floating to other integer formats.

— The translation time conversion of floating constants and the **strtod**, **strtof**, **strtold**, **fprintf**, **fscanf**, and related library functions in **<stdlib.h>**, **<stdio.h>**, and **<wchar.h>** provide IEC 60559 binary-decimal conversions. The **strtold** function in **<stdlib.h>** provides the conv function recommended in the Appendix to ANSI/IEEE 854.

— The relational and equality operators provide IEC 60559 comparisons. IEC 60559 identifies a need for additional comparison predicates to facilitate writing code that accounts for NaNs. The comparison macros (**isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, and **isunordered**) in **<math.h>** supplement the language operators to address this need. The **islessgreater** and **isunordered** macros provide respectively a quiet version of the <> predicate and the unordered predicate recommended in the Appendix to IEC 60559.

— The **feclearexcept**, **feraiseexcept**, and **fetestexcept** functions in **<fenv.h>** provide the facility to test and alter the IEC 60559 floating-point exception status flags. The **fegetexceptflag** and **fesetexceptflag** functions in **<fenv.h>** provide the facility to save and restore all five status flags at one time. These functions are used in conjunction with the type **fexcept_t** and the floating-point exception macros (**FE_INEXACT**, **FE_DIVBYZERO**, **FE_UNDERFLOW**, **FE_OVERFLOW**, **FE_INVALID**) also in **<fenv.h>**.

— The **fegetround** and **fesetround** functions in **<fenv.h>** provide the facility to select among the IEC 60559 directed rounding modes represented by the rounding direction macros in **<fenv.h>** (**FE_TONEAREST**, **FE_UPWARD**, **FE_DOWNWARD**, **FE_TOWARDZERO**) and the values **0**, **1**, **2**, and **3** of **FLT_ROUNDS** are the IEC 60559 directed rounding modes.

— The **fegetenv**, **feholdexcept**, **fesetenv**, and **feupdateenv** functions in **<fenv.h>** provide a facility to manage the floating-point environment, comprising the IEC 60559 status flags and control modes.

— The **copysign** functions in **<math.h>** provide the copysign function recommended in the Appendix to IEC 60559.

— The **fabs** functions in **<math.h>** provide the abs function recommended in the Appendix to IEC 60559.

— The unary minus (**-**) operator provides the unary minus (-) operation recommended in the Appendix to IEC 60559.

— The **scalbn** and **scalbln** functions in **<math.h>** provide the scalb function recommended in the Appendix to IEC 60559.

— The **logb** functions in **<math.h>** provide the logb function recommended in the Appendix to IEC 60559, but following the newer specifications in ANSI/IEEE 854.

— The **nextafter** and **nexttoward** functions in **<math.h>** provide the nextafter function recommended in the Appendix to IEC 60559 (but with a minor change to better handle signed zeros).

— The **isfinite** macro in **<math.h>** provides the finite function recommended in the Appendix to IEC 60559.

— The **isnan** macro in **<math.h>** provides the isnan function recommended in the Appendix to IEC 60559.

— The **signbit** macro and the **fpclassify** macro in **<math.h>**, used in conjunction with the number classification macros (**FP_NAN**, **FP_INFINITE**, **FP_NORMAL**, **FP_SUBNORMAL**, **FP_ZERO**), provide the facility of the class function recommended in the Appendix to IEC 60559 (except that the classification macros defined in 7.12.3 do not distinguish signaling from quiet NaNs).

with:

F.3 Operations

[1] C operators, functions, and function-like macros provide the operations required by IEC 60559 as shown in the following table. Specifications for the C facilities are provided in the listed clauses. The C specifications are intended to match IEC 60559, unless stated otherwise.

Operation binding

IEC 60559 operation	C operation	Clauses - C11
roundToIntegralTiesToEven	`roundeven`	7.12.9.7a, F.10.6.7a
roundToIntegralTiesAway	`round`	7.12.9.6, F.10.6.6
roundToIntegralTowardZero	`trunc`	7.12.9.8, F.10.6.8
roundToIntegralTowardPositive	`ceil`	7.12.9.1, F.10.6.1
roundToIntegralTowardNegative	`floor`	7.12.9.2, F.10.6.2
roundToIntegralExact	`rint`	7.12.9.4, F.10.6.4
nextUp	`nextup`	7.12.11.5, F.10.8.5
nextDown	`nextdown`	7.12.11.6, F.10.8.6
remainder	`remainder`, `remquo`	7.12.10.2, F.10.7.2, 7.12.10.3, F.10.7.3
minNum	`fmin`	7.12.12.3, F.10.9.3
maxNum	`fmax`	7.12.12.2, F.10.9.2
minNumMag	`fminmag`	7.12.12.5, F.10.9.5
maxNumMag	`fmaxmag`	7.12.12.4, F.10.9.4
scaleB	`scalbn`, `scalbln`	7.12.6.13, F.10.3.13
logB	`logb`, `ilogb`, `llogb`	7.12.6.11, F.10.3.11, 7.12.6.5, F.10.3.5 7.12.6.6a, F.10.3.6a
addition	`+`, `fadd`, `faddl`, `daddl`	6.5.6, 7.12.13a.1, F.10.10a
subtraction	`-`, `fsub`, `fsubl`, `dsubl`	6.5.6, 7.12.13a.2, F.10.10a
multiplication	`*`, `fmul`, `fmull`, `dmull`	6.5.5, 7.12.13a.3, F.10.10a
division	`/`, `fdiv`, `fdivl`, `ddivl`	6.5.5, 7.12.13a.4, F.10.10a
squareRoot	`sqrt`, `fsqrt`, `fsqrtl`, `dsqrtl`	7.12.13a.6, F.10.10a
fusedMultiplyAdd	`fma`, `ffma`, `ffmal`, `dfmal`	7.12.13.1, F.10.10.1, 7.12.13a.5, F.10.10a
convertFromInt	cast and implicit conversion	6.3.1.4, 6.5.4
convertToIntegerTiesToEven	`fromfp`, `ufromfp`	7.12.9.9, F.10.6.9
convertToIntegerTowardZero	`fromfp`, `ufromfp`	7.12.9.9, F.10.6.9
convertToIntegerTowardPositive	`fromfp`, `ufromfp`	7.12.9.9, F.10.6.9
convertToIntegerTowardNegative	`fromfp`, `ufromfp`	7.12.9.9, F.10.6.9
convertToIntegerTiesToAway	`fromfp`, `ufromfp`, `lround`, `llround`	7.12.9.9, F.10.6.9, 7.12.9.7, F.10.6.7
convertToIntegerExactTiesToEven	`fromfpx`, `ufromfpx`	7.12.9.10, F.10.6.10
convertToIntegerExactTowardZero	`fromfpx`, `ufromfpx`	7.12.9.10, F.10.6.10

convertToIntegerExactTowardPositive	**fromfpx**, **ufromfpx**	7.12.9.10, F.10.6.10
convertToIntegerExactTowardNegative	**fromfpx**, **ufromfpx**	7.12.9.10, F.10.6.10
convertToIntegerExactTiesToAway	**fromfpx**, **ufromfpx**	7.12.9.10, F.10.6.10
convertFormat - different formats	cast and implicit conversions	6.3.1.5, 6.5.4
convertFormat - same format	**canonicalize**	7.12.11.7, F.10.8.7
convertFromDecimalCharacter	**strtod**, **wcstod**, **scanf**, **wscanf**, decimal floating constants	7.22.1.3, 7.29.4.1.1, 7.21.6.2, 7.29.2.12, F.5
convertToDecimalCharacter	**printf**, **wprintf**, **strfromd**, **strfromf**, **strfroml**	7.21.6.1, 7.29.2.11, 7.22.1.2a, F.5
convertFromHexCharacter	**strtod**, **wcstod**, **scanf**, **wscanf**, hexadecimal floating constants	7.22.1.3, 7.29.4.1.1, 7.21.6.2, 7.29.2.12, F.5
convertToHexCharacter	**printf**, **wprintf**, **strfromd**, **strfromf**, **strfroml**	7.21.6.1, 7.29.2.11, 7.22.1.2a, F.5
copy	**memcpy**, **memmove**	7.24.2.1, 7.24.2.2
negate	**-(x)**	6.5.3.3
abs	**fabs**	7.12.7.2, F.10.4.2
copySign	**copysign**	7.12.11.1, F.10.8.1
compareQuietEqual	**==**	6.5.9, F.9.3
compareQuietNotEqual	**!=**	6.5.9, F.9.3
compareSignalingEqual	**iseqsig**	7.12.14.7, F.10.11.1
compareSignalingGreater	**>**	6.5.8, F.9.3
compareSignalingGreaterEqual	**>=**	6.5.8, F.9.3
compareSignalingLess	**<**	6.5.8, F.9.3
compareSignalingLessEqual	**<=**	6.5.8, F.9.3
compareSignalingNotEqual	**! iseqsig(x)**	7.12.14.7, F.10.11.1
compareSignalingNotGreater	**! (x > y)**	6.5.8, F.9.3
compareSignalingLessUnordered	**! (x >= y)**	6.5.8, F.9.3
compareSignalingNotLess	**! (x < y)**	6.5.8, F.9.3
compareSignalingGreaterUnordered	**! (x <= y)**	6.5.8, F.9.3
compareQuietGreater	**isgreater**	7.12.14.1
compareQuietGreaterEqual	**isgreaterequal**	7.12.14.2
compareQuietLess	**isless**	7.12.14.3
compareQuietLessEqual	**islessequal**	7.12.14.4
compareQuietUnordered	**isunordered**	7.12.14.6
compareQuietNotGreater	**! isgreater(x, y)**	7.12.14.1
compareQuietLessUnordered	**! isgreaterequal(x, y)**	7.12.14.2
compareQuietNotLess	**! isless(x, y)**	7.12.14.3
compareQuietGreaterUnordered	**! islessequal(x, y)**	7.12.14.4
compareQuietOrdered	**! isunordered(x, y)**	7.12.14.6
class	**fpclassify**, **signbit**, **issignaling**	7.12.3.1, 7.12.3.6, 7.12.3.7
isSignMinus	**signbit**	7.12.3.6
isNormal	**isnormal**	7.12.3.5
isFinite	**isfinite**	7.12.3.2
isZero	**iszero**	7.12.3.9

isSubnormal	**issubnormal**	7.12.3.8
isInfinite	**isinf**	7.12.3.3
isNaN	**isnan**	7.12.3.4
isSignaling	**issignaling**	7.12.3.7
isCanonical	**iscanonical**	7.12.3.1a
radix	**FLT_RADIX**	5.2.4.2.2
totalOrder	**totalorder**	F.10.12.1
totalOrderMag	**totalordermag**	F.10.12.2
lowerFlags	**feclearexcept**	7.6.2.1
raiseFlags	**fesetexcept**	7.6.2.3a
testFlags	**fetestexcept**	7.6.2.5
testSavedFlags	**fetestexceptflag**	7.6.2.4a
restoreFlags	**fesetexceptflag**	7.6.2.4
saveAllFlags	**fegetexceptflag**	7.6.2.2
getBinaryRoundingDirection	**fegetround**	7.6.3.1
setBinaryRoundingDirection	**fesetround**	7.6.3.2
saveModes	**fegetmode**	7.6.3.0
restoreModes	**fesetmode**	7.6.3.1a
defaultModes	**fesetmode(FE_DFL_MODE)**	7.6.3.1a, 7.6

[2] The IEC 60559 requirement that certain of its operations be provided for operands of different formats (of the same radix) is satisfied by C's usual arithmetic conversions (6.3.1.8) and function-call argument conversions (6.5.2.2). For example, the following operations take **float f** and **double d** inputs and produce a **long double** result:

```
(long double)f * d
powl(f, d)
```

[3] Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation is implementation-defined, even if **<fenv.h>** defines the macro **FE_SNANS_ALWAYS_SIGNAL** (F.2.1). If the return expression of a **return** statement is evaluated to the floating-point format of the return type, it is implementation-defined whether a convertFormat operation is applied to the result of the return expression.

[4] The unary **-** operator raises no floating-point exceptions, even if the operand is a signaling NaN.

[5] The C classification macros **fpclassify**, **iscanonical**, **isfinite**, **isinf**, **isnan**, **isnormal**, **issignaling**, **issubnormal**, and **iszero** provide the IEC 60559 operations indicated in the table above provided their arguments are in the format of their semantic type. Then these macros raise no floating-point exceptions, even if an argument is a signaling NaN.

[6] The C **nearbyint** functions (7.12.9.3, F.10.6.3) provide the nearbyinteger function recommended in the Appendix to (superseded) ANSI/IEEE 854.

[7] The C **nextafter** (7.12.11.3, F.10.8.3) and **nexttoward** (7.12.11.4, F.10.8.4) functions provide the nextafter function recommended in the Appendix to (superseded) IEC 60559:1989 (but with a minor change to better handle signed zeros).

[8] The C **getpayload**, **setpayload**, and **setpayloadsig** (F.10.13) functions provide program access to NaN payloads, defined in IEC 60559.

[9] The macros (7.6) **FE_DOWNWARD**, **FE_TONEAREST**, **FE_TOWARDZERO**, and **FE_UPWARD**, which are used in conjunction with the **fegetround** and **fesetround** functions and the **FENV_ROUND** pragma, represent the IEC 60559 rounding-direction attributes roundTowardNegative, roundTiesToEven, roundTowardZero, and roundTowardPositive, respectively.

[10] The C **fegetenv** (7.6.4.1), **feholdexcept** (7.6.4.2), **fesetenv** (7.6.4.3) and **feupdateenv** (7.6.4.4) functions provide a facility to manage the dynamic floating-point environment, comprising the IEC 60559 status flags and dynamic control modes.

[11] IEC 60559 requires operations with specified operand and result formats. Therefore, math functions that are bound to IEC 60559 operations (see table above) must remove any extra range and precision from arguments or results.

[12] IEC 60559 requires operations that round their result to formats the same as and wider than the operands, in addition to the operations that round their result to narrower formats (see 7.12.13a). Operators (**+**, **-**, *****, and **/**) whose evaluation formats are wider than the semantic type (5.2.4.2.2) might not support some of the IEEE 60559 operations, because getting a result in a given format might require a cast that could introduce an extra rounding error. The functions that round result to narrower type (7.12.13a) provide the IEC 60559 operations that round result to same and wider (as well as narrower) formats, in those cases where built-in operators and casts do not. For example, **ddivl(x, y)** computes a correctly rounded **double** divide of **float x** by **float y**, regardless of the evaluation method.

In F.3#3, attach a footnote to the wording:

Whether C assignment (6.5.16) (and conversion as if by assignment) to the same format is an IEC 60559 convertFormat or copy operation

where the footnote is:

*) Where the source and destination formats are the same, convertFormat operations differ from copy operations in that convertFormat operations raise the "invalid" floating-point exception on signaling NaN inputs and do not propagate non-canonical encodings.

9 Floating to integer conversion

IEC 60559 allows but does not require floating to integer type conversions to raise the "inexact" floating-point exception for non-integer inputs within the range of the integer type. It recommends that implicit conversions raise "inexact" in these cases.

Change to C11:

Replace footnote 360):

360) ANSI/IEEE 854, but not IEC 60559 (ANSI/IEEE 754), directly specifies that floating-to-integer conversions raise the "inexact" floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the "inexact" floating-point exception. See **rint**, **lrint**, **llrint**, and **nearbyint** in **<math.h>**.

with:

360) IEC 60559 recommends that implicit floating-to-integer conversions raise the "inexact" floating-point exception for non-integer in-range values. In those cases where it matters, library functions can be used to effect such conversions with or without raising the "inexact" floating-point exception. See **fromfp**, **ufromfp**, **fromfpx**, **ufromfpx**, **rint**, **lrint**, **llrint**, and **nearbyint** in **<math.h>**.

10 Conversions between floating types and character sequences

10.1 Conversions with decimal character sequences

IEC 60559 now requires correct rounding for conversions between its supported formats and decimal character sequences with up to H decimal digits, where H is defined as follows:

$H \quad \geq M + 3$

$M \quad = 1+\text{ceiling}(p\times\log_{10}(2))$

p is the precision of the widest supported IEC 60559 binary format

M is large enough that conversion from the widest supported format to a decimal character sequence with M decimal digits and back will be the identity function. IEC 60559 also now completely specifies conversions involving more than H decimal digits. The following changes to C11 satisfy these requirements.

Changes to C11:

Rename F.5 from:

F.5 Binary-decimal conversion

to:

F.5 Conversions between binary floating types and decimal character sequences

After F.5#2, insert:

[2a] The **<float.h>** header defines the macro

CR_DECIMAL_DIG

if and only if **__STDC_WANT_IEC_60559_BFP_EXT__** is defined as a macro at the point in the source file where **<float.h>** is first included. If defined, **CR_DECIMAL_DIG** expands to an integral constant expression suitable for use in **#if** preprocessing directives whose value is a number such that conversions between all supported types with IEC 60559 binary formats and character sequences with at most **CR_DECIMAL_DIG** significant decimal digits are correctly rounded. The value of **CR_DECIMAL_DIG** shall be at least **DECIMAL_DIG** + 3. If the implementation correctly rounds for all numbers of significant decimal digits, then **CR_DECIMAL_DIG** shall have the value of the macro **UINTMAX_MAX**.

[2b] Conversions of types with IEC 60559 binary formats to character sequences with more than **CR_DECIMAL_DIG** significant decimal digits shall correctly round to **CR_DECIMAL_DIG** significant digits and pad zeros on the right.

[2c] Conversions from character sequences with more than **CR_DECIMAL_DIG** significant decimal digits to types with IEC 60559 binary formats shall correctly round to an intermediate character sequence with **CR_DECIMAL_DIG** significant decimal digits, according to the applicable rounding direction, and correctly round the intermediate result (having **CR_DECIMAL_DIG** significant decimal digits) to the destination type. The "inexact" floating-point exception is raised (once) if either conversion is inexact. (The second conversion may raise the "overflow" or "underflow" floating-point exception.)

In F.5#2c, attach a footnote to the wording:

> The "inexact" floating-point exception is raised (once) if either conversion is inexact.

where the footnote is:

> *) The intermediate conversion is exact only if all input digits after the first **CR_DECIMAL_DIG** digits are **0.**

In 5.2.4.2.2#7, change:

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

to:

> All except **CR_DECIMAL_DIG** (F.5), **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types.

10.2 Conversions to character sequences

The following change to C11 allows freestanding implementations to provide the conversions from floating types to character sequences as required by IEC 60559, without having to support **<stdio.h>**.

Change to C11:

After 7.22.1.2, add:

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`**.

11 Constant rounding directions

IEC 60559 now requires a means for programs to specify constant values for the rounding direction mode for all standard operations in static parts of code (as specified by the programming language). The following changes meet this requirement by adding standard pragmas for specifying constant values for the rounding direction mode. Minor terminology changes in the C11 references to rounding direction modes and the floating-point environment are needed to distinguish two kinds of rounding direction modes: constant and dynamic.

Changes to C11:

Change 5.1.2.3#5:

[5] When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type **`volatile sig_atomic_t`** are unspecified, as is the state of the floating-point environment. The value of any object that is modified by the handler that is neither a lock-free atomic object nor of type **`volatile sig_atomic_t`** becomes indeterminate when the handler exits, as does the state of the floating-point environment if it is modified by the handler and not restored.

to:

[5] When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type **`volatile sig_atomic_t`** are unspecified, as is the state of the dynamic floating-point environment. The value of any object that is modified by the handler that is neither a lock-free atomic object nor of type **`volatile sig_atomic_t`** becomes indeterminate when the handler exits, as does the state of the dynamic floating-point environment if it is modified by the handler and not restored.

After 7.6#1, insert the paragraph:

[1a] A floating-point control mode may be *constant* (7.6.2) or *dynamic*. The *dynamic floating-point environment* includes the dynamic floating-point control modes and the floating-point status flags.

Replace 7.6#2:

> [2] The floating-point environment has thread storage duration. The initial state for a thread’s floating-point environment is the current state of the floating-point environment of the thread that creates it at the time of creation.

with:

> [2] The dynamic floating-point environment has thread storage duration. The initial state for a thread’s dynamic floating-point environment is the current state of the dynamic floating-point environment of the thread that creates it at the time of creation.

Replace 7.6#3:

> [3] Certain programming conventions support the intended model of use for the floating-point environment: ...

with:

> [3] Certain programming conventions support the intended model of use for the dynamic floating-point environment: ...

Replace 7.6#4:

> [4] The type
>
> **`fenv_t`**
>
> represents the entire floating-point environment.

with:

> [4] The type
>
> **`fenv_t`**
>
> represents the entire dynamic floating-point environment.

Replace 7.6#9:

> [9] The macro
>
> **`FE_DFL_ENV`**
>
> represents the default floating-point environment — the one installed at program startup — and has type “pointer to const-qualified **`fenv_t`**”. It can be used as an argument to **`<fenv.h>`** functions that manage the floating-point environment.

with:

[9] The macro

FE_DFL_ENV

represents the default dynamic floating-point environment — the one installed at program startup — and has type "pointer to const-qualified **fenv_t**". It can be used as an argument to **<fenv.h>** functions that manage the dynamic floating-point environment.

Modify 7.6.1#2 by replacing:

If part of a program tests floating-point status flags, sets floating-point control modes, or runs under non-default mode settings, but was translated with the state for the **FENV_ACCESS** pragma "off", the behavior is undefined.

with:

If part of a program tests floating-point status flags or establishes non-default floating-point mode settings using any means other than the **FENV_ROUND** pragmas, but was translated with the state for the **FENV_ACCESS** pragma "off", the behavior is undefined.

Modify footnote 213) by replacing:

In general, if the state of **FENV_ACCESS** is "off", the translator can assume that default modes are in effect and the flags are not tested.

with:

In general, if the state of **FENV_ACCESS** is "off", the translator can assume that the flags are not tested, and that default modes are in effect, except where specified otherwise by an **FENV_ROUND** pragma.

Following 7.6.1 "The **FENV_ACCESS** pragma", insert:

7.6.1a Rounding control pragma

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <fenv.h>
#pragma STDC FENV_ROUND direction
```

Description

[2] The **FENV_ROUND** pragma provides a means to specify a constant rounding direction for floating-point operations within a translation unit or compound statement. The pragma shall occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another **FENV_ROUND** pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another **FENV_ROUND** pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound

statement the static rounding mode is restored to its condition just before the compound statement. If this pragma is used in any other context, its behavior is undefined.

[3] ***direction*** shall be one of the rounding direction macro names defined in 7.6, or **FE_DYNAMIC.** If any other value is specified, the behavior is undefined. If no **FENV_ROUND** pragma is in effect, or the specified constant rounding mode is **FE_DYNAMIC,** rounding is according to the mode specified by the dynamic floating-point environment, which is the dynamic rounding mode that was established either at thread creation or by a call to **fesetround, fesetmode, fesetenv,** or **feupdateenv.** If the **FE_DYNAMIC** mode is specified and **FENV_ACCESS** is "off", the translator may assume that the default rounding mode is in effect.

[4] Within the scope of an **FENV_ROUND** directive establishing a mode other than **FE_DYNAMIC,** all floating-point operators, implicit conversions (including the conversion of a value represented in a format wider than its semantic types to its semantic type, as done by classification macros), and invocations of functions indicated in the table below, for which macro replacement has not been suppressed (7.1.4), shall be evaluated according to the specified constant rounding mode (as though no constant mode was specified and the corresponding dynamic rounding mode had been established by a call to **fesetround**). Invocations of functions for which macro replacement has been suppressed and invocations of functions other than those indicated in the table below shall not be affected by constant rounding modes — they are affected by (and affect) only the dynamic mode. Floating constants (6.4.4.2) that occur in the scope of a constant rounding mode shall be interpreted according to that mode.

Functions affected by constant rounding modes

Header	Function groups
<math.h>	**acos**, **asin**, **atan**, **atan2**
<math.h>	**cos**, **sin**, **tan**
<math.h>	**acosh**, **asinh**, **atanh**
<math.h>	**cosh**, **sinh**, **tanh**
<math.h>	**exp**, **exp2**, **expm1**
<math.h>	**log**, **log10**, **log1p**, **log2**
<math.h>	**scalbn**, **scalbln**, **ldexp**
<math.h>	**cbrt**, **hypot**, **pow**, **sqrt**
<math.h>	**erf**, **erfc**
<math.h>	**lgamma**, **tgamma**
<math.h>	**rint**, **nearbyint**, **lrint**, **llrint**
<math.h>	**fdim**
<math.h>	**fma**
<math.h>	**fadd**, **daddl**, **fsub**, **dsubl**, **fmul**, **dmull**, **fdiv**, **ddivl**, **ffma**, **dfmal**, **fsqrt**, **dsqrtl**
<stdlib.h>	**atof**, **strfromd**, **strfromf**, **strfroml**, **strtod**, **strtof**, **strtold**
<wchar.h>	**wcstod**, **wcstof**, **wcstold**
<stdio.h>	**printf** and **scanf** families
<wchar.h>	**wprintf** and **wscanf** families

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

[5] Constant rounding modes (other than **FE_DYNAMIC**) could be implemented using dynamic rounding modes as illustrated in the following example:

```
{
   #pragma STDC FENV_ROUND direction
   // compiler inserts:
   // #pragma STDC FENV_ACCESS ON
   // int __savedrnd;
   // __savedrnd = __swapround(direction);
   ... operations affected by constant rounding mode ...
   // compiler inserts:
   // __savedrnd = __swapround(__savedrnd);
   ... operations not affected by constant rounding mode ...
   // compiler inserts:
   // __savedrnd = __swapround(__savedrnd);
   ... operations affected by constant rounding mode ...
   // compiler inserts:
   // __swapround(__savedrnd);
}
```

where **__swapround** is defined by:

```
static inline int __swapround(const int new) {
   const int old = fegetround();
   fesetround(new);
   return old;
}
```

In 7.6.3.1#2, change:

[2] The **fegetround** function gets the current rounding direction.

to:

[2] The **fegetround** function gets the current value of the dynamic rounding direction mode.

In 7.6.3.1#3, change:

[3] The **fegetround** function returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

to:

[3] The **fegetround** function returns the value of the rounding direction macro representing the current dynamic rounding direction or a negative value if there is no such rounding direction macro or the current dynamic rounding direction is not determinable.

In 7.6.3.2#2, change:

> [2] The **fesetround** function establishes the rounding direction represented by its argument **round**. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

to:

> [2] The **fesetround** function sets the dynamic rounding direction mode to the rounding direction represented by its argument **round**. If the argument is not equal to the value of a rounding direction macro, the rounding direction is not changed.

In 7.6.3.2#3, change:

> [3] The **fesetround** function returns zero if and only if the requested rounding direction was established.

to:

> [3] The **fesetround** function returns zero if and only if the dynamic rounding direction mode was set to the requested rounding direction.

In 7.6.4.1 Description, change:

> [2] The **fegetenv** function attempts to store the current floating-point environment in the object pointed to by **envp**.

to:

> [2] The **fegetenv** function attempts to store the current dynamic floating-point environment in the object pointed to by **envp**.

In 7.6.4.2 Description, change:

> [2] The **feholdexcept** function saves the current floating-point environment in the object pointed to by **envp**

to:

> [2] The **feholdexcept** function saves the current dynamic floating-point environment in the object pointed to by **envp**

In 7.6.4.3 Description, change:

> [2] The **fesetenv** function attempts to establish the floating-point environment represented by the object pointed to by **envp**. The argument **envp** shall point to an object set by a call to **fegetenv** or **feholdexcept**, or equal a floating-point environment macro.

to:

> [2] The **fesetenv** function attempts to establish the dynamic floating-point environment represented by the object pointed to by **envp**. The argument **envp** shall point to an object set by a call to **fegetenv** or **feholdexcept**, or equal a dynamic floating-point environment macro.

In 7.6.4.4 Description, change:

> [2] The **feupdateenv** function attempts to save the currently raised floating-point exceptions in its automatic storage, install the floating-point environment represented by the object pointed to by **envp**, and then raise the saved floating-point exceptions. The argument **envp** shall point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a floating-point environment macro.

to:

> [2] The **feupdateenv** function attempts to save the currently raised floating-point exceptions in its automatic storage, install the dynamic floating-point environment represented by the object pointed to by **envp**, and then raise the saved floating-point exceptions. The argument **envp** shall point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a dynamic floating-point environment macro.

In F.8.1, replace:

> [1] IEC 60559 requires that floating-point operations implicitly raise floating-point exception status flags, and that rounding control modes can be set explicitly to affect result values of floating-point operations. When the state for the **FENV_ACCESS** pragma (defined in **<fenv.h>**) is "on", these changes to the floating-point state are treated as side effects which respect sequence points.364)

with:

> [1] IEC 60559 requires that floating-point operations implicitly raise floating-point exception status flags, and that rounding control modes can be set explicitly to affect result values of floating-point operations. These changes to the floating-point state are treated as side effects which respect sequence points.364)

Change footnote 364) from:

> 364) If the state for the **FENV_ACCESS** pragma is "off", the implementation is free to assume the floating-point control modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see F.9).

to:

> 364) If the state for the **FENV_ACCESS** pragma is "off", the implementation is free to assume the dynamic floating-point control modes will be the default ones and the floating-point status flags will not be tested, which allows certain optimizations (see F.9).

In F.8.2, replace:

> [1] During translation the IEC 60559 default modes are in effect:

with:

> [1] During translation, constant rounding direction modes (7.6.2) are in effect where specified. Elsewhere, during translation the IEC 60559 default modes are in effect:

Change footnote 365) from:

> 365) As floating constants are converted to appropriate internal representations at translation time, their conversion is subject to default rounding modes and raises no execution-time floating-point exceptions (even where the state of the **FENV_ACCESS** pragma is "on"). Library functions, for example **strtod**, provide execution-time conversion of numeric strings.

to:

> 365) As floating constants are converted to appropriate internal representations at translation time, their conversion is subject to constant or default rounding modes and raises no execution-time floating-point exceptions (even where the state of the **FENV_ACCESS** pragma is "on"). Library functions, for example **strtod**, provide execution-time conversion of numeric strings.

In F.8.3, replace:

> [1] At program startup the floating-point environment is initialized ...

with:

> [1] At program startup the dynamic floating-point environment is initialized ...

In F.8.3, change the second bullet from:

> — The rounding direction mode is rounding to nearest.

to:

> — The dynamic rounding direction mode is rounding to nearest.

12 NaN support

The 2011 update to IEC 60559 retains support for signaling NaNs. Although C11 notes that floating types may contain signaling NaNs, it does not otherwise specify signaling NaNs. Some unqualified references to NaNs in C11 do not properly apply to signaling NaNs, so that an implementation could not add signaling NaN support as an extension without contradicting C11. The goal of the following changes is to allow implementations to conditionally support signaling NaNs as specified in IEC 60559, but to require only minimal support for signaling NaNs.

Changes to C11:

In 7.12.1#2, after the second sentence, insert:

> Whether a signaling NaN input causes a domain error is implementation-defined.

After 7.12#5, add:

> [5a] The signaling NaN macros
>
> **SNANF**
> **SNAN**
> **SNANL**

each is defined if and only if the respective type contains signaling NaNs (5.2.4.2.2). They expand to a constant expression of the respective 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.

In 7.12.14, change 4th sentence from:

> The following subclauses provide macros that are *quiet* (non floating-point exception raising) versions of the relational operators, and other comparison macros that facilitate writing efficient code that accounts for NaNs without suffering the ‘‘invalid’’ floating-point exception.

to:

> Subclauses 7.12.14.1 through 7.12.14.6 provide macros that are quiet versions of the relational operators: the macros do not raise the "invalid" floating-point exception as an effect of quiet NaN arguments. The comparison macros facilitate writing efficient code that accounts for quiet NaNs without suffering the ‘‘invalid’’ floating-point exception.

In the second paragraphs of 7.12.14.1 through 7.12.14.5, append to "when **x** and **y** are unordered" the phrase "and neither is a signaling NaN".

In 7.12.14.6#2, append to the Description: "The **isunordered** macro raises no floating-point exceptions if neither argument is a signaling NaN."

Change F.2.1 from:

> **F.2.1 Infinities, signed zeros, and NaNs**
>
> [1] This specification does not define the behavior of signaling NaNs.359~~342~~) It generally uses the term *NaN* to denote quiet NaNs. The **NAN** and **INFINITY** macros and the **nan** functions in **<math.h>** provide designations for IEC 60559 NaNs and infinities.

to:

> **F.2.1 Infinities and NaNs**
>
> [1] Since negative and positive infinity are representable in IEC 60559 formats, all real numbers lie within the range of representable values (5.2.4.2.2).
>
> [2] The **NAN** and **INFINITY** macros and the **nan** functions in **<math.h>** provide designations for IEC 60559 quiet NaNs and infinities. The **SNANF**, **SNAN**, and **SNANL** macros in **<math.h>** provide designations for IEC 60559 signaling NaNs.
>
> [3] This annex does not require the full support for signaling NaNs specified in IEC 60559. This annex uses the term *NaN*, unless explicitly qualified, to denote quiet NaNs. Where specification of signaling NaNs is not provided, the behavior of signaling NaNs is implementation-defined (either treated as an IEC 60559 quiet NaN or treated as an IEC 60559 signaling NaN).359)
>
> [4] Any operator or **<math.h>** function that raises an "invalid" floating-point exception, if delivering a floating type result, shall return a quiet NaN.
>
> [5] In order to support signaling NaNs as specified in IEC 60559, an implementation should adhere to the following recommended practice.

Recommended practice

[6] Any floating-point operator or **<math.h>** function or macro with a signaling NaN input, unless explicitly specified otherwise, raises an "invalid" floating-point exception.

[7] NOTE Some functions do not propagate quiet NaN arguments. For example, **hypot(x, y)** returns infinity if **x** or **y** is infinite and the other is a quiet NaN. The recommended practice in this subclause specifies that such functions (and others) raise the "invalid" floating-point exception if an argument is a signaling NaN, which also implies they return a quiet NaN in these cases.

[8] The **<fenv.h>** header defines the macro

FE_SNANS_ALWAYS_SIGNAL

if and only if the implementation follows the recommended practice in this subclause. If defined, **FE_SNANS_ALWAYS_SIGNAL** expands to the integer constant **1**.

In F.4, change the first sentence from:

If the integer type is **_Bool**, 6.3.1.2 applies and no floating-point exceptions are raised (even for NaN).

to:

If the integer type is **_Bool**, 6.3.1.2 applies and the conversion raises no floating-point exceptions if the floating-point value is not a signaling NaN.

Append to the end of F.5 the following paragraph:

[4] The **fprintf** family of functions in **<stdio.h>** and the **fwprintf** family of functions in **<wchar.h>** should behave as if floating-point operands were passed through the **canonicalize** function of the same type.

In F.5#4, attach a footnote to the wording:

The **fprintf** family of functions in **<stdio.h>** and the **fwprintf** family of functions in **<wchar.h>** should behave as if floating-point operands were passed through the **canonicalize** function of the same type.

where the footnote is:

*) This is a recommendation instead of a requirement so that implementations may choose to print signaling NaNs differently from quiet NaNs.

In F.9.2, bullet 1*x and x/1 -> x, replace "are equivalent" with "may be regarded as equivalent".

In F.10#3, change the last sentence:

The other functions in **<math.h>** treat infinities, NaNs, signed zeros, subnormals, and (provided the state of the **FENV_ACCESS** pragma is "on") the floating-point status flags in a manner consistent with the basic arithmetic operations covered by IEC 60559.

to:

> The other functions in **<math.h>** treat infinities, NaNs, signed zeros, subnormals, and (provided the state of the **FENV_ACCESS** pragma is “on”) the floating-point status flags in a manner consistent with IEC 60559 operations.

After F.10#4, insert:

> [4a] The functions bound to operations in IEC 60559 (F.3) are fully specified by IEC 60559, including rounding behaviors and floating-point exceptions.

In F.10, replace paragraphs 8 through 10:

> [8] Whether or when library functions raise the “inexact” floating-point exception is unspecified, unless explicitly specified otherwise.
>
> [9] Whether or when library functions raise an undeserved “underflow” floating-point exception is unspecified.372) Otherwise, as implied by F.8.6, the **<math.h>** functions do not raise spurious floating-point exceptions (detectable by the user), other than the “inexact” floating-point exception.
>
> [10] Whether the functions honor the rounding direction mode is implementation-defined, unless explicitly specified otherwise.

with:

> [8] Whether or when library functions not bound to operations in IEC 60559 raise the “inexact” floating-point exception is unspecified, unless stated otherwise.
>
> [9] Whether or when library functions not bound to operations in IEC 60559 raise an undeserved “underflow” floating-point exception is unspecified.372) Otherwise, as implied by F.8.6, these functions do not raise spurious floating-point exceptions (detectable by the user), other than the “inexact” floating-point exception.
>
> [10] Whether the functions not bound to operations in IEC 60559 honor the rounding direction mode is implementation-defined, unless explicitly specified otherwise.

Append to footnote 374):

> Note also that this implementation does not handle signaling NaNs as required of implementations that define **FP_SNANS_ALWAYS_SIGNAL**.

Change footnotes 242) and 243) from:

> 242) NaN arguments are treated as missing data: if one argument is a NaN and the other numeric, then the **fmax** functions choose the numeric value. See F.10.9.2.
>
> 243) The **fmin** functions are analogous to the **fmax** functions in their treatment of NaNs.

to:

> 242) Quiet NaN arguments are treated as missing data: if one argument is a quiet NaN and the other numeric, then the **fmax** functions choose the numeric value. See F.10.9.2.

243) The **fmin** functions are analogous to the **fmax** functions in their treatment of quiet NaNs.

In F.10.3.4, replace paragraphs 2 and 3:

[2] **frexp** raises no floating-point exceptions.

[3] When the radix of the argument is a power of 2, the returned value is exact and is independent of the current rounding direction mode.

with:

[2] **frexp** raises no floating-point exceptions if **value** is not a signaling NaN.

[3] The returned value is independent of the current rounding direction mode.

In F.10.4.2, replace paragraph 2:

[2] The returned value is exact and is independent of the current rounding direction mode.

with:

[2] **fabs(***x***)** raises no floating-point exceptions, even if *x* is a signaling NaN. The returned value is independent of the current rounding direction mode.

In F.10.4.5, replace paragraph 1:

[1] **sqrt** is fully specified as a basic arithmetic operation in IEC 60559. The returned value is dependent on the current rounding direction mode.

with:

— **sqrt(**±0**)** returns ±0.

— **sqrt(**+∞**)** returns +∞.

— **sqrt(***x***)** returns a NaN and raises the "invalid" floating-point exception for $x < 0$.

The returned value is dependent on the current rounding direction mode.

In F.10.6.6#3, attach a footnote to the wording:

The **double** version of round behaves as though implemented by

where the footnote is:

*) This code does not handle signaling NaNs as required of implementations that define **FP_SNANS_ALWAYS_SIGNAL**.

In F.10.7.2, replace paragraph 1:

[1] The **remainder** functions are fully specified as a basic arithmetic operation in IEC 60559.

with:

— `remainder(±0, y)` returns ±0 for *y* not zero.

— `remainder(`*x*`, `*y*`)` returns a NaN and raises the “invalid” floating-point exception for *x* infinite or *y* zero (and neither is a NaN).

— `remainder(`*x*`, ±∞)` returns *x* for *x not infinite.*

In F.10.8.1, replace paragraph 2:

[2] The returned value is exact and is independent of the current rounding direction mode.

with:

[2] `copysign(`*x*`, `*y*`)` raises no floating-point exceptions, even if *x* or *y* is a signaling NaN. The returned value is independent of the current rounding direction mode.

In F.10.9.2, paragraph 3, change the sample implementation for **`fmax`** from:

```
{ return (isgreaterequal(x, y) ||
                    isnan(y)) ? x : y; }
```

to:

```
{
   double r;
   r = (isgreaterequal(x, y) || isnan(y)) ? x : y;
   (void) canonicalize(&r, &r);
   return r;
}
```

In G.3#1, replace:

[1] A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a NaN). ...

with:

[1] A complex or imaginary value with at least one infinite part is regarded as an *infinity* (even if its other part is a quiet NaN). ...

After G.6#4, append the paragraph:

[4a] In subsequent subclauses in G.6 "NaN" refers to a quiet NaN. The behavior of signaling NaNs in Annex G is implementation-defined.

Change footnote 378) from:

378) As noted in G.3, a complex value with at least one infinite part is regarded as an infinity even if its other part is a NaN.

to:

> 378) As noted in G.3, a complex value with at least one infinite part is regarded as an infinity even if its other part is a quiet NaN.

13 Integer width macros

C11 clause 6.2.6.2 defines the *width* of integer types. These widths are needed in order to use the **fromfp**, **ufromfp**, **fromfpx**, and **ufromfpx** functions to round to the integer types. The following changes to C11 provide macros for the widths of integer types. On the belief that width macros would be generally useful, the proposal adds them to **<limits.h>** and **<stdint.h>**.

Changes to C11:

In 5.2.4.2.1#1, change:

> Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

to:

> Moreover, except for **CHAR_BIT, MB_LEN_MAX**, and the width-of-type macros, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

In 5.2.4.2.1#1, insert the following bullets, each after the current bullets for the same type:

— width of type **char**
  CHAR_WIDTH 8
— width of type **signed char**
  SCHAR_WIDTH 8
— width of type **unsigned char**
  UCHAR_WIDTH 8
— width of type **short int**
  SHRT_WIDTH 16
— width of type **unsigned short int**
  USHRT_WIDTH 16
— width of type **int**
  INT_WIDTH 16
— width of type **unsigned int**
  UINT_WIDTH 16
— width of type **long int**
  LONG_WIDTH 32
— width of type **unsigned long int**
  ULONG_WIDTH 32
— width of type **long long int**
  LLONG_WIDTH 64
— width of type **unsigned long long int**
  ULLONG_WIDTH 64

In 7.20.2#2, change:

> Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives, and this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

to:

> Each instance of any defined macro shall be replaced by a constant expression suitable for use in **#if** preprocessing directives, and, except for the width-of-type macros, this expression shall have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions.

In 7.20.2.1, append:

— width of exact-width signed integer types
INT*N***_WIDTH** *N*
— width of exact-width unsigned integer types
UINT*N***_WIDTH** *N*

In 7.20.2.2, append:

— width of minimum-width signed integer types
INT_LEAST*N***_WIDTH** *N*
— width of minimum-width unsigned integer types
UINT_LEAST*N***_WIDTH** *N*

In 7.20.2.3, append:

— width of fastest minimum-width signed integer types
INT_FAST*N***_WIDTH***N*
— width of fastest minimum-width unsigned integer types
UINT_FAST*N***_WIDTH** *N*

In 7.20.2.4, append:

— width of pointer-holding signed integer type
INTPTR_WIDTH 16
— width of pointer-holding unsigned integer type
UINTPTR_WIDTH 16

In 7.20.2.5, append:

— width of greatest-width signed integer type
INTMAX_WIDTH 64
— width of greatest-width unsigned integer type
UINTMAX_WIDTH 64

In 7.20.3#2, insert the following macros, each after the current macros for the same type:

PTRDIFF_WIDTH 16
SIG_ATOMIC_WIDTH 8
SIZE_WIDTH 16

```
WCHAR_WIDTH  8
WINT_WIDTH   16
```

14 Mathematics `<math.h>`

The 2011 update to IEC 60559 requires several new operations that are appropriate for **`<math.h>`**. Also, in a few cases, it tightens requirements for functions that are already in C11 **`<math.h>`**.

14.0 C underflow

The following change to C11 loosens the C definition of underflow to encompass IEC 60559 gradual underflow (see C11 footnote 232).

Changes to C11:

Change the first sentence in 7.12.1#6 from:

> [6] The result underflows if the magnitude of the mathematical result is so small that the mathematical result cannot be represented, without extraordinary roundoff error, in an object of the specified type.232) ...

to:

> [6] The result underflows if the magnitude of the mathematical result is nonzero and less than the minimum normal number in the type.232) ...

14.1 Nearest integer functions

14.1.1 Round to integer value in floating type

IEC 60559 requires a function that rounds a value of floating type to an integer value in the same floating type, without raising the “inexact” floating-point exception, for each of the rounding methods: to nearest, to nearest even, upward, downward, and toward zero. The C11 **`round`**, **`ceil`**, **`floor`**, and **`trunc`** functions may meet this requirement for four of the five rounding methods, though are permitted to raise the “inexact” floating-point exception. The following changes add a function that rounds to nearest and remove the latitude to raise the “inexact” floating-point exception.

Changes to C11:

Change F.10.6.1:

> [2] The returned value is independent of the current rounding direction mode.

to:

> [2] The returned value is exact and is independent of the current rounding direction mode.

In F.10.6.1#3, change:

```
result = rint(x); // or nearbyint instead of rint
```

to:

```
result = nearbyint(x);
```

Delete F.10.6.1#4:

> The **ceil** functions may, but are not required to, raise the "inexact" floating-point exception for finite non-integer arguments, as this implementation does.

Change F.10.6.2:

> [2] The returned value is independent of the current rounding direction mode.

to:

> [2] The returned value is exact and is independent of the current rounding direction mode.

Delete the second sentence of F.10.6.2#3:

> The **floor** functions may, but are not required to, raise the "inexact" floating-point exception for finite non-integer arguments, as that implementation does.

Change F.10.6.6:

> [2] The returned value is independent of the current rounding direction mode.

to:

> [2] The returned value is exact and is independent of the current rounding direction mode.

Change F.10.6.6#3 from:

> [3] The **double** version of **round** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double round(double x)
{
   double result;
   fenv_t save_env;
   feholdexcept(&save_env);
   result = rint(x);
   if (fetestexcept(FE_INEXACT)) {
         fesetround(FE_TOWARDZERO);
         result = rint(copysign(0.5 + fabs(x), x));
   }
   feupdateenv(&save_env);
   return result;
}
```

> The **round** functions may, but are not required to, raise the "inexact" floating-point exception for finite non-integer numeric arguments, as this implementation does.

to:

> [3] The **double** version of **round** behaves as though implemented by

```
#include <math.h>
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
double round(double x)
{
   double result;
   fenv_t save_env;
   feholdexcept(&save_env);
   result = rint(x);
   if (fetestexcept(FE_INEXACT)) {
      fesetround(FE_TOWARDZERO);
      result = rint(copysign(0.5 + fabs(x), x));
      feclearexcept(FE_INEXACT);
   }
   feupdateenv(&save_env);
   return result;
}
```

After 7.12.9.7, add:

7.12.9.7a The **roundeven** functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double roundeven(double x);
float roundevenf(float x);
long double roundevenl(long double x);
```

Description

[2] The **roundeven** functions round their argument to the nearest integer value in floating-point format, rounding halfway cases to even (that is, to the nearest value that is an even integer~~that is, to the nearest value whose least significant bit 0~~), regardless of the current rounding direction.

Returns

[3] The **roundeven** functions return the rounded integer value.

After F.10.6.7, add:

F.10.6.7a The **roundeven** functions

[1]
— **roundeven(±0)** returns ±0.
— **roundeven(±∞)** returns ±∞.

[2] The returned value is exact and is independent of the current rounding direction mode.

[3] See the sample implementation for **ceil** in F.10.6.1.

In F.10.6.8#1, delete the second sentence: The returned value is exact.

Replace F.10.6.8#2:

> [2] The returned value is independent of the current rounding direction mode. The **`trunc`** functions may, but are not required to, raise the "inexact" floating-point exception for finite non-integer arguments.

with:

> [2] The returned value is exact and is independent of the current rounding direction mode.

14.1.2 Convert to integer type

IEC 60559 requires conversion operations from each of its formats to each integer format, signed and unsigned, for each of five different rounding methods. For each of these it requires an operation that raises the "inexact" floating-point exception (for non-integer in-range inputs) and an operation that does not raise the "inexact" floating-point exception. The changes below satisfy this requirement with four new functions that take two extra arguments to represent the rounding direction and the rounding precision.

Changes to C11:

After 7.12#6, add:

[6a] The math rounding direction macros

```
FP_INT_UPWARD
FP_INT_DOWNWARD
FP_INT_TOWARDZERO
FP_INT_TONEARESTFROMZERO
FP_INT_TONEAREST
```

represent the rounding directions of the functions **`ceil`**, **`floor`**, **`trunc`**, **`round`**, and **`roundeven`**, respectively, that convert to integral values in floating-point formats. They expand to integer constant expressions with distinct values suitable for use as the second argument to the **`fromfp`**, **`ufromfp`**, **`fromfpx`**, and **`ufromfpx`** functions.

After 7.12.9.8, add:

7.12.9.9 The `fromfp` and `ufromfp` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdint.h>
#include <math.h>
intmax_t fromfp(double x, int round, unsigned int width);
intmax_t fromfpf(float x, int round, unsigned int width);
intmax_t fromfpl(long double x, int round, unsigned int width);
uintmax_t ufromfp(double x, int round, unsigned int width);
uintmax_t ufromfpf(float x, int round, unsigned int width);
uintmax_t ufromfpl(long double x, int round, unsigned int width);
```

Description

[2] The **fromfp** and **ufromfp** functions round **x**, using the math rounding direction indicated by **round**, to a signed or unsigned integer, respectively, of **width** bits, and return the result value in the integer type designated by **intmax_t** or **uintmax_t**, respectively. If the value of the **round** argument is not equal to the value of a math rounding direction macro, the direction of rounding is unspecified. If the value of **width** exceeds the width of the function type, the rounding is to the full width of the function type. The **fromfp** and **ufromfp** functions do not raise the "inexact" floating-point exception. If **x** is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width, or if **width** is zero, the functions return an unspecified value and a domain error occurs.

Returns

[3] The **fromfp** and **ufromfp** functions return the rounded integer value.

[4] EXAMPLE Upward rounding of double **x** to type **int**, without raising the "inexact" floating-point exception, is achieved by

```
(int)fromfp(x, FP_INT_UPWARD, INT_WIDTH)
```

7.12.9.10 The fromfpx and ufromfpx functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <stdint.h>
#include <math.h>
intmax_t fromfpx(double x, int round, unsigned int width);
intmax_t fromfpxf(float x, int round, unsigned int width);
intmax_t fromfpxl(long double x, int round, unsigned int width);
uintmax_t ufromfpx(double x, int round, unsigned int width);
uintmax_t ufromfpxf(float x, int round, unsigned int width);
uintmax_t ufromfpxl(long double x, int round, unsigned int width);
```

Description

[2] The **fromfpx** and **ufromfpx** functions differ from the **fromfp** and **ufromfp** functions, respectively, only in that the **fromfpx** and **ufromfpx** functions raise the "inexact" floating-point exception if a rounded result not exceeding the specified width differs in value from the argument **x**.

Returns

[3] The **fromfpx** and **ufromfpx** functions return the rounded integer value.

[4] NOTE Conversions to integer types that are not required to raise the inexact exception can be done simply by rounding to integral value in floating type and then converting to the target integer type. For example, the conversion of **long double x** to **uint64_t**, using upward rounding, is done by

```
(uint64_t)ceill(x)
```

In 7.12.9.9#2, attach a footnote to the wording:

any supported integer type

where the footnote is:

*) For signed types, 6.2.6.2 permits three representations, which differ in whether a value of -(2^M), where M is the number of value bits, can be represented.

After F.10.6.8, add:

F.10.6.9 The `fromfp` and `ufromfp` functions

[1] The **`fromfp`** and **`ufromfp`** functions raise the “invalid” floating-point exception and return an unspecified value if the floating-point argument **`x`** is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width.

[2] These functions do not raise the “inexact” floating-point exception.

F.10.6.10 The `fromfpx` and `ufromfpx` functions

[1] The **`fromfpx`** and **`ufromfpx`** functions raise the “invalid” floating-point exception and return an unspecified value if the floating-point argument **`x`** is infinite or NaN or rounds to an integral value that is outside the range of any supported integer type of the specified width.

[2] These functions raise the “inexact” floating-point exception if a valid result differs in value from the floating-point argument **`x`**.

14.2 The `llogb` functions

IEC 60559 requires that its logB operations, for invalid input, return a value outside ±2 × (*emax* + *p* ~~-~~– 1), where *emax* is the maximum exponent and *p* the precision of the floating-point input format. If the width of the **`int`** type is only 16 bits and the floating type has a 15-bit exponent (like the binary128 format), then the **`ilogb`** functions cannot meet this requirement. The following changes to C11 add the **`llogb`** functions, which return **`long int`** and hence can satisfy this requirement for the **`long double`** types provided by current and expected implementations.

Changes to C11:

After 7.12#8, add:

[8.a] The macros

```
FP_LLOGB0
FP_LLOGBNAN
```

expand to integer constant expressions whose values are returned by **`llogb(x)`** if **`x`** is zero or NaN, respectively. The value of **`FP_LLOGB0`** shall be **`LONG_MIN`** if the value of **`FP_LOGB0`** is **`INT_MIN`**, and shall be **`-LONG_MAX`** if the value of **`FP_LOGB0`** is **`−INT_MAX`**. The value of **`FP_LLOGBNAN`** shall be **`LONG_MAX`** if the value of **`FP_LOGBNAN`** is **`INT_MAX`**, and shall be **`LONG_MIN`** if the value of **`FP_LOGBNAN`** is **`INT_MIN`**.

After 7.12.6.6, add:

7.12.6.6a The `llogb` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
long int llogb(double x);
long int llogbf(float x);
long int llogbl(long double x);
```

Description

[2] The **llogb** functions extract the exponent of **x** as a signed **long int** value. If **x** is zero they compute the value **FP_LLOGB0**; if **x** is infinite they compute the value **LONG_MAX**; if **x** is a NaN they compute the value **FP_LLOGBNAN**; otherwise, they are equivalent to calling the corresponding **logb** function and casting the returned value to type **long int**. A domain error or range error may occur if **x** is zero, infinite, or NaN. If the correct value is outside the range of the return type, the numeric result is unspecified.

Returns

[3] The **llogb** functions return the exponent of **x** as a signed **long int** value.

Forward references: the **logb** functions (7.12.6.11).

After F.10.3.6, add:

F.10.3.6a The `llogb` functions

[1] The **llogb** functions are equivalent to the **ilogb** functions, except that the **llogb** functions determine a result in the **long int** type.

14.3 Max-min magnitude functions

IEC 60559 requires functions that determine which of two inputs has the maximum and minimum magnitude.

Changes to C11:

After 7.12.12.3, add:

7.12.12.4 The `fmaxmag` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double fmaxmag(double x, double y);
float fmaxmagf(float x, float y);
long double fmaxmagl(long double x, long double y);
```

Description

[2] The **fmaxmag** functions determine the value of their argument whose magnitude is the maximum of the magnitudes of the arguments: the value of **x** if |**x**| > |**y**|, **y** if |**x**| < |**y**|, and **fmax(x, y)** otherwise.

Returns

[3] The **fmaxmag** functions return the value of their argument of maximum magnitude.

7.12.12.5 The fminmag functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double fminmag(double x, double y);
float fminmagf(float x, float y);
long double fminmagl(long double x, long double y);
```

Description

[2] The **fminmag** functions determine the value of their argument whose magnitude is the minimum of the magnitudes of the arguments: the value of **x** if |**x**| < |**y**|, **y** if |**x**| > |**y**|, and **fmin(x, y)** otherwise.

Returns

[3] The **fminmag** functions return the value of their argument of minimum magnitude.

In 7.12.12.4#2, attach a footnote to the wording:

the value of **x** if |**x**| > |**y**|, **y** if |**x**| < |**y**|, and **fmax(x, y)** otherwise.

where the footnote is:

*) Quiet NaN arguments are treated as missing data: if one argument is a quiet NaN and the other numeric, then the **fmaxmag** functions choose the numeric value. See F.10.9.4.

In 7.12.12.5#2, attach a footnote to the wording:

the value of **x** if |**x**| < |**y**|, **y** if |**x**| > |**y**|, and **fmin(x, y)** otherwise.

where the footnote is:

*) The **fminmag** functions are analogous to the **fmaxmag** functions in their treatment of quiet NaNs.

After F.10.9.3, add:

F.10.9.4 The fmaxmag functions

[1] If just one argument is a NaN, the **fmaxmag** functions return the other argument (if both arguments are NaNs, the functions return a NaN).

[2] The returned value is exact and is independent of the current rounding direction mode.

[3] The body of the **fmaxmag** function might be

```
{
   double ax, ay, r;
   ax = fabs(x);
   ay = fabs(y);
   if (isgreater(ax, ay)) (void)canonicalize(&r, &x);
   else if (isgreater(ay, ax)) (void)canonicalize(&r, &y);
   else r = fmax(x, y);
   return r;
}
```

F.10.9.5 The fminmag functions

[1] The **fminmag** functions are analogous to the **fmaxmag** functions (F.10.9.4).

[2] The returned value is exact and is independent of the current rounding direction mode.

14.4 The nextup and nextdown functions

IEC 60559 replaces the previously recommended two-argument nextAfter operation with one-argument nextUp and nextDown operations. C11 supports the nextAfter operation with the **nextafter** and **nexttoward** functions. The following changes to C11 add functions for the new operations and retain the **nextafter** and **nexttoward** functions already in C11.

Changes to C11:

After 7.12.11.4 add:

7.12.11.5 The nextup functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double nextup(double x);
float nextupf(float x);
long double nextupl(long double x);
```

Description

[2] The **nextup** functions determine the next representable value, in the type of the function, greater than **x**. If **x** is the negative number of least magnitude in the type of **x**, **nextup(x)** is −0 if the type has signed zeros and is 0 otherwise. If **x** is zero, **nextup(x)** is the positive number of least magnitude in the type of **x**. **nextup(HUGE_VAL)** is **HUGE_VAL**.

Returns

[3] The **nextup** functions return the next representable value in the specified type greater than **x**.

7.12.11.6 The `nextdown` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double nextdown(double x);
float nextdownf(float x);
long double nextdownl(long double x);
```

Description

[2] The **nextdown** functions determine the next representable value, in the type of the function, less than **x**. If **x** is the positive number of least magnitude in the type of **x**, **nextdown(x)** is +0 if the type has signed zeros and is 0 otherwise. If **x** is zero, **nextdown(x)** is the negative number of least magnitude in the type of **x**. **nextdown(-HUGE_VAL)** is ~~–~~ −~~-~~**HUGE_VAL**.

Returns

[3] The **nextdown** functions return the next representable value in the specified type less than **x**.

After F.10.8.4, add:

F.10.8.5 The `nextup` functions

[1]

— **nextup(+∞)** returns +∞.

— **nextup(−∞)** returns the largest-magnitude negative finite number in the type of the function.

nextup(x) raises no floating-point exceptions if **x** is not a signaling NaN. The returned value is independent of the current rounding direction mode.

F.10.8.6 The nextdown functions

[1]

— **nextdown(+∞)** returns the largest-magnitude positive finite number in the type of the function.

— **nextdown(−∞)** returns −∞.

nextdown(x) raises no floating-point exceptions if **x** is not a signaling NaN. The returned value is independent of the current rounding direction mode.

14.5 Functions that round result to narrower type

IEC 60559 requires add, subtract, multiply, divide, fused multiply-add, and square root operations that round once to a floating-point format independent of the format of the operands. The following changes to C11 add functions for these operations that round to formats narrower than the operand formats.

Changes to C11:

After 7.12#7, add:

[7a] Each of the macros

```
FP_FAST_FADD
FP_FAST_FADDL
FP_FAST_DADDL
FP_FAST_FSUB
FP_FAST_FSUBL
FP_FAST_DSUBL
FP_FAST_FMUL
FP_FAST_FMULL
FP_FAST_DMULL
FP_FAST_FDIV
FP_FAST_FDIVL
FP_FAST_DDIVL
FP_FAST_FSQRT
FP_FAST_FSQRTL
FP_FAST_DSQRTL
```

is optionally defined. If defined, it indicates that the corresponding function generally executes about as fast, or faster, than the corresponding operation of the argument type (with result type the same as the argument type) followed by conversion to the narrower type. (For **FP_FAST_FFMA**, **FP_FAST_FFMAL**, and **FP_FAST_DFMAL**, the comparison is to a call to **fma** or **fmal** followed by a conversion, not to separate multiply, add, and conversion.) If defined, these macros expand to the integer constant 1.

After 7.12.13, add:

7.12.13a Functions that round result to narrower type

[1] The functions in this subclause round their results to a type typically narrower than the parameter types.

7.12.13a.1 Add and round to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
float fadd(double x, double y);
float faddl(long double x, long double y);
double daddl(long double x, long double y);
```

Description

[2] These functions compute the sum **x** + **y**, rounded to the type of the function. They compute the sum (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error may occur for infinite arguments.

Returns

[3] These functions return the sum **x** + **y**, rounded to the type of the function.

7.12.13a.2 Subtract and round to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
float fsub(double x, double y);
float fsubl(long double x, long double y);
double dsubl(long double x, long double y);
```

Description

[2] These functions compute the difference **x** − **y**, rounded to the type of the function. They compute the difference (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error may occur for infinite arguments.

Returns

[3] These functions return the difference **x** − **y**, rounded to the type of the function.

7.12.13a.3 Multiply and round to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
float fmul(double x, double y);
float fmull(long double x, long double y);
double dmull(long double x, long double y);
```

Description

[2] These functions compute the product **x** × **y**, rounded to the type of the function. They compute the product (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error occurs for one infinite argument and one zero argument.

Returns

[3] These functions return the product of **x** × **y**, rounded to the type of the function.

7.12.13a.4 Divide and round to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
float fdiv(double x, double y);
float fdivl(long double x, long double y);
double ddivl(long double x, long double y);
```

Description

[2] These functions compute the quotient **x** ÷ **y**, rounded to the type of the function. They compute the quotient (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error occurs for either both arguments infinite or both arguments zero. A pole error occurs for a finite x and a zero y.

Returns

[3] These functions return the quotient **x** ÷ **y**, rounded to the type of the function.

7.12.13a.5 Floating multiply-add rounded to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
float ffma(double x, double y, double z);
float ffmal(long double x, long double y, long double z);
double dfmal(long double x, long double y, long double z);
```

Description

[2] These functions compute (**x** × **y**) + **z**, rounded to the type of the function. They compute (**x** ~~×~~ × **y**) + **z** (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite arguments. A domain error may occur for an infinite argument.

Returns

[3] These functions return (**x** × **y**) + **z**, rounded to the type of the function.

7.12.13a.6 Square root rounded to narrower type

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
float fsqrt(double x);
float fsqrtl(long double x);
double dsqrtl(long double x);
```

Description

[2] These functions compute the square root of **x**, rounded to the type of the function. They compute the square root (as if) to infinite precision and round once to the result format, according to the current rounding mode. A range error may occur for finite positive arguments. A domain error occurs if the argument is less than zero.

Returns

[3] These functions return the square root of **x**, rounded to the type of the function.

In 7.12.13a #1, attach a footnote to the wording:

typically narrower

where the footnote is:

*) In some cases the destination type might not be narrower than the parameter types. For example, **double** might not be narrower than **long double**.

After F.10.10 add:

F.10.10a Functions that round result to narrower type

[1] The functions that round their result to narrower type (7.12.13a) are fully specified in IEC 60559. The returned value is dependent on the current rounding direction mode.

[2] These functions treat zero and infinite arguments like the corresponding operation or function: **+**, **-**, *****, **/**, **fma**, or **sqrt**.

14.6 Comparison macros

IEC 60559 requires an extensive set of comparison operations. C11's built-in equality and relational operators and quiet comparison macros and their negations (**!**) support all these required operations, except for compareSignalingEqual and compareSignalingNotEqual. The following changes to C11 provide a function-like macro for compareSignalingEqual. The negation of the macro provides compareSignalingNotEqual.

Changes to C11:

After 7.12.14.6, add:

7.12.14.7 The iseqsig macro

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int iseqsig(real-floating x, real-floating y);
```

Description

[2] The **iseqsig** macro determines whether its arguments are equal. If an argument is a NaN, a domain error occurs for the macro, as if a domain error occurred for a function (7.12.1).

Returns

[3] The **iseqsig** macro returns 1 if its arguments are equal and 0 otherwise.

After F.10.11, add:

F.10.11.1 The iseqsig macro

[1] The equality operator **==** and the **iseqsig** macro produce equivalent results, except that the **iseqsig** macro raises the "invalid" floating-point exception if an argument is a NaN.

14.7 Classification macros

IEC 60559 requires several classification operations, all but four of which are already supported in C11 as function-like macros. The changes to C11 below support the remaining four.

Changes to C11:

After 7.12.3.1, add:

7.12.3.1a The iscanonical macro

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int iscanonical(real-floating~~real-floating~~ x);
```

Description

[2] The **iscanonical** macro determines whether its argument value is canonical (5.2.4.2.2). First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

[3] The **iscanonical** macro returns a nonzero value if and only if its argument is canonical.

~~At the end of~~After 7.12.3.6, add:

7.12.3.7 The issignaling macro

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int issignaling(real-floating~~real-floating~~ x);
```

Description

[2] The **issignaling** macro determines whether its argument value is a signaling NaN.

Returns

[3] The **`issignaling`** macro returns a nonzero value if and only if its argument is a signaling NaN.

7.12.3.8 The `issubnormal` macro

Synopsis

[1] **`#define __STDC_WANT_IEC_60559_BFP_EXT__`**
`#include <math.h>`
`int issubnormal(`*real-floating* ~~**`real-floating`**~~ **`x);`**

Description

[2] The **`issubnormal`** macro determines whether its argument value is subnormal. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

[3] The **`issubnormal`** macro returns a nonzero value if and only if its argument is subnormal.

7.12.3.9 The `iszero` macro

Synopsis

[1] **`#define __STDC_WANT_IEC_60559_BFP_EXT__`**
`#include <math.h>`
`int iszero(`*real-floating* ~~**`real-floating`**~~ **`x);`**

Description

[2] The **`iszero`** macro determines whether its argument value is (positive, negative, or unsigned) zero. First, an argument represented in a format wider than its semantic type is converted to its semantic type. Then determination is based on the type of the argument.

Returns

[3] The **`iszero`** macro returns a nonzero value if and only if its argument is zero.

In 7.12.3.7#2, attach a footnote to the wording:

The **`issignaling`** macro determines whether its argument value is a signaling NaN.

where the footnote is:

*) F.3 specifies that **`issignaling`** (and all the other classification macros), raise no floating-point exception if the argument is a variable, or any other expression whose value is represented in the format of the its semantic type, even if the value is a signaling NaN.

14.8 Total order functions

IEC 60559 requires a totalOrder operation, which it defines as follows:

"totalOrder(x, *y)* imposes a total ordering on canonical members of the format of *x* and *y*:

a) If $x < y$, totalOrder(x, *y)* is true.
b) If $x > y$, totalOrder(x, *y)* is false.
c) If $x = y$:
   1) totalOrder(−0, +0) is true.
   2) totalOrder(+0, −0) is false.
   3) If *x* and *y* represent the same floating-point datum:
      i) If *x* and *y* have negative sign, totalOrder(x, *y)* is true if and only if the exponent of $x \geq$ the exponent of *y*
      ii) otherwise totalOrder(x, *y)* is true if and only if the exponent of $x \leq$ the exponent of *y.*
d) If *x* and *y* are unordered numerically because *x* or *y* is NaN:
   1) totalOrder(−NaN, *y)* is true where −NaN represents a NaN with negative sign bit and *y* is a floating-point number.
   2) totalOrder(x, +NaN) is true where +NaN represents a NaN with positive sign bit and *x* is a floating-point number.
   3) If *x* and *y* are both NaNs, then totalOrder reflects a total ordering based on:
      i) negative sign orders below positive sign
      ii) signaling orders below quiet for +NaN, reverse for −NaN
      iii) lesser payload, when regarded as an integer, orders below greater payload for +NaN, reverse for −NaN."

IEC 60559:2011 also requires a totalOrderMag operation which is the totalOrder of the absolute values of the operands. The following changes to C11 provide these operations.

Changes to C11:

After F.10.11, add:

F.10.12 Total order functions

[1] This ~~annex~~subclause specifies the total order functions required by IEC 60559.

F.10.12.1 The `totalorder` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int totalorder(double x, double y);
int totalorderf(float x, float y);
int totalorderl(long double x, long double y);
```

Description

[2] The **`totalorder`** functions determine whether the total order relationship, defined by IEC 60559, is true for the ordered pair of its arguments **`x`**, **`y`**. These functions are fully specified in IEC 60559. These functions are independent of the current rounding direction mode and raise no floating-point exceptions, even if an argument is a signaling NaN.

Returns

[3] The **`totalorder`** functions return nonzero if and only if the total order relation is true for the ordered pair of its arguments **`x`**, **`y`**.

F.10.12.2 The `totalordermag` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
int totalordermag(double x, double y);
int totalordermagf(float x, float y);
int totalordermagl(long double x, long double y);
```

Description

[2] The **`totalordermag`** functions determine whether the total order relationship, defined by IEC 60559, is true for the ordered pair of the magnitudes of its arguments **`x`**, **`y`**. These functions are fully specified in IEC 60559. These functions are independent of the current rounding direction mode and raise no floating-point exceptions, even if an argument is a signaling NaN.

Returns

[3] The **`totalordermag`** functions return nonzero if and only if the total order relation is true for the ordered pair of the magnitudes of it arguments **`x`**, **`y`**.

In F.10.12#1, attach a footnote to the wording:

These functions are fully specified in IEC 60559.

where the footnote is:

*) The total order functions are specified only in Annex F because they depend on the details of IEC 60559 formats.

14.9 Canonicalize functions

IEC 60559 requires an arithmetic convertFormat operation from each format to itself. This operation produces a canonical encoding and, for a signaling NaN input, raises the “invalid” floating-point exception and delivers a quiet NaN. C assignment (and conversion as if by assignment) to the same format may be implemented as a convertFormat operation or as a copy operation. The changes to C11 below provide the IEC 60559 convertFormat operation.

Changes to C11:

As the last subclause of 7.12.11 (after 7.12.11.5-6 added above), add:

7.12.11.7 The `canonicalize` functions

Synopsis

[1]

```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int canonicalize(double * cx, const double * x);
int canonicalizef(float * cx, const float * x);
int canonicalizel(long double * cx, const long double * x);
```

Description

[2] The **`canonicalize`** functions attempt to produce a canonical version of the floating-point representation in the object pointed to by the argument **`x`**, as if to a temporary object of the specified type, and store the canonical result in the object pointed to by the argument **`cx`**. If the input **`*x`** is a signaling NaN, the **`canonicalize`** functions are intended to store a canonical quiet NaN. If a canonical result is not produced the object pointed to by **`cx`** in unchanged.

Returns

[3] The functions return zero if a canonical result is stored in the object pointed to by **`cx`**. Otherwise they return a nonzero value.

In 7.12.11.7#2, attach a footnote to the wording:

> and store the canonical result in the object pointed to by the argument **`cx`**.

where the footnote is:

> *) Arguments **`x`** and **`cx`** may point to the same object.

After F.10.8.6 (added above), add:

F.10.8.7 The `canonicalize` functions

[1] The **`canonicalize`** functions produce the canonical version of the representation in the object pointed to by the argument **`x`**. If the input **`*x`** is a signaling NaN, the "invalid" floating-point exception is raised and a (canonical) quiet NaN (which should be the canonical version of that signaling NaN made quiet) is produced. For quiet NaN, infinity, and finite inputs, the functions raise no floating-point exceptions.

In F.10.8.7#1, attach a footnote to the wording:

> The **`canonicalize`** functions produce

where the footnote is:

> *) As if **`*x * 1e0`** were computed.

14.10 NaN functions

IEC 60559 defines the payload of a NaN to be a certain part of the NaN's significand interpreted as an integer. The payload is intended to provide implementation-defined diagnostic information about the NaN, such as where or how the NaN was created. The following change to C11 provides functions to get and set the NaN payloads defined in IEC 60559.

Change to C11:

After F.10.12 (added above), add:

F.10.13 Payload functions

IEC 60559 defines the *payload* of a quiet or signaling NaN as information encoded in part of the NaN significand. The payload can be interpreted as an unsigned integer.~~IEC 60559 defines the *payload* of a quiet or signaling NaN as an integer value encoded in the significand.~~ The payload is intended to represent implementation-defined diagnostic information about the NaN. The functions in this clause enable getting and setting payloads.

F.10.13.1 The `getpayload` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
double getpayload(const double *x);
float getpayloadf(const float *x);
long double getpayloadl(const long double *x);
```

Description

[2] The **`getpayload`** functions extract the integer value of the payload of a NaN input and return the integer as a floating-point value. The sign of the returned integer is positive. If **`*x`** is not a NaN, the return result is unspecified. These functions raise no floating-point exceptions, even if **`*x`** is a signaling NaN.

Returns

[3] The functions return a floating-point representation of the integer value of the payload of the NaN input.

F.10.13.2 The `setpayload` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int setpayload(double *res, double pl);
int setpayloadf(float *res, float pl);
int setpayloadl(long double *res, long double pl);
```

Description

[2] The **`setpayload`** functions create a quiet NaN with the payload specified by **`pl`** and a zero sign bit and store that NaN in the object pointed to by **`*res`**. If **`pl`** is not a floating-point integer representing a valid payload, **`*res`** is set to positive zero.~~If **`pl`** is not a positive floating-point integer representing a valid payload, **`*res`** is set to positive zero.~~

Returns

[3] If the functions stored the specified NaN, the functions return a zero value, otherwise a non-zero value (and **`*res`** is set to zero).

F.10.13.3 The `setpayloadsig` functions

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <math.h>
int setpayloadsig(double *res, double pl);
int setpayloadsigf(float *res, float pl);
int setpayloadsigl(long double *res, long double pl);
```

Description

[2] The **`setpayloadsig`** functions create a signaling NaN with the payload specified by **`pl`** and a zero sign bit and store that NaN in the object pointed to by **`*res`**. If **`pl`** is not a floating-point integer representing a valid payload, **`*res`** is set to positive zero.~~If **`pl`** is not a positive floating-point integer representing a valid payload, **`*res`** is set to positive zero.~~

Returns

[3] If the functions stored the specified NaN, the functions return a zero value, otherwise a non-zero value (and **`*res`** is set to zero).

15 The floating-point environment `<fenv.h>`

15.1 The `fesetexcept` function

IEC 60559 requires a raiseFlags operation that sets floating-point exception flags. Unlike the C **`feraiseexcept`** function in **`<fenv.h>`**, the raiseFlags operation does not cause side effects (notably traps) as could occur if the exceptions resulted from arithmetic operations. The following changes to C11 provide the raiseFlags operation.

Changes to C11:

After 7.6.2.3, add:

7.6.2.3a The `fesetexcept` function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <fenv.h>
int fesetexcept(int excepts);
```

Description

[2] The **`fesetexcept`** function attempts to set the supported floating-point exception flags represented by its argument. This function does not clear any floating-point exception flags. This function changes the state of the floating-point exception flags, but does not cause any other side effects that might be associated with raising floating-point exceptions.

Returns

[3] The **`fesetexcept`** function returns zero if all the specified exceptions were successfully set or if the **`excepts`** argument is zero. Otherwise, it returns a nonzero value.

In 7.6.2.3a#2, attach a footnote to the wording:

but does not cause any other side effects that might be associated with raising floating-point exceptions.

where the footnote is:

*) Enabled traps for floating-point exceptions are not taken.

15.2 The `fetestexceptflag` function

IEC 60559 requires a testSavedFlags operation to test saved representations of floating-point exception flags. This differs from the C **`fetestexcept`** function in **`<fenv.h>`** which tests floating-point exception flags directly. The following change to C11 provides the testSavedFlags operation.

Change to C11:

After 7.6.2.4, add:

7.6.2.4a The `fetestexceptflag` function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <fenv.h>
int fetestexceptflag(const fexcept_t * flagp, int excepts);
```

Description

[2] The **`fetestexceptflag`** determines which of a specified subset of the floating-point exception flags are set in the object pointed to by **`flagp`**. The value of **`*flagp`** shall have been set by a previous call to **`fegetexceptflag`** whose second argument represented at least those floating-point exceptions represented by the argument **`excepts`**.~~The value of **`*flagp`** shall have been set by a previous call to **`fegetexceptflag`**.~~ The **`excepts`** argument specifies the floating-point status flags to be queried.

Returns

[3] The ~~**`fetestexcept`**~~**`fetestexceptflag`** function returns the value of the bitwise OR of the floating-point exception macros included in **excepts** corresponding to the floating-point exceptions set in **`*flagp`**.

15.3 Control modes

IEC 60559 requires a saveModes operation that saves all the user-specifiable dynamic floating-point modes supported by the implementation, including dynamic rounding direction and trap enablement modes. The following changes to C11 support this operation.

Changes to C11:

After 7.6#5, add:

[5a] The type

`femode_t`

represents the collection of dynamic floating-point control modes supported by the implementation, including the dynamic rounding direction mode.

After 7.6#7, add:

[7a] The macro

`FE_DFL_MODE`

represents the default state for the collection of dynamic floating-point control modes supported by the implementation - and has type "pointer to const-qualified **`femode_t`**". Additional implementation-defined states for the dynamic mode collection, with macro definitions beginning with **`FE_`** and an uppercase letter, and having type "pointer to const-qualified **`femode_t`**", may also be specified by the implementation.

Rename 7.6.3 from:

7.6.3 Rounding

to:

7.6.3 Rounding and other control modes

Append to 7.6.3#1:

The **`fegetmode`** and **`fesetmode`** functions manage all the implementation's dynamic floating-point control modes collectively.

Before 7.6.3.1, insert:

7.6.3.0 The `fegetmode` function

Synopsis

[1]
```
#define __STDC_WANT_IEC_60559_BFP_EXT__
#include <fenv.h>
int fegetmode(femode_t *modep);
```

Description

[2] The **fegetmode** function attempts to store all the dynamic floating-point control modes in the object pointed to by **modep**.

Returns

[3] The **fegetmode** function returns zero if the modes were successfully stored. Otherwise, it returns a nonzero value.

After 7.6.3.1, add:

7.6.3.1a The fesetmode function

Synopsis

[1] **#define __STDC_WANT_IEC_60559_BFP_EXT__**
#include <fenv.h>
int fesetmode(const ~~fenv~~femode_t *modep);

Description

[2] The **fesetmode** function attempts to establish the dynamic floating-point modes represented by the object pointed to by **modep.** The argument **modep** shall point to an object set by a call to **fegetmode,** or equal **FE_DFL_MODE** or a dynamic floating-point mode state macro defined by the implementation.

Returns

[3] The **fesetmode** function returns zero if the modes were successfully established. Otherwise, it returns a nonzero value.

16 Type-generic math <tgmath.h>

The following changes to C11 enhance the specification for type-generic math macros to accommodate functions and the constant rounding mode pragma in this Part of Technical Specification 18661.

<tgmath.h> is not intended to define type-generic macros associated with functions that have not been declared for lack of a defined **__STDC_WANT_IEC_60559_BFP_EXT__** macro.

Changes to C11:

In 7.25#2, change:

For each such function, except **modf,** there is a corresponding *type-generic macro*.

to:

For each such function, except **modf**, **setpayload**, and **setpayloadsig**, there is a corresponding *type-generic macro*.

In 7.25#3, replace:

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

with:

[3] Except for the macros for functions that round result to a narrower type (7.12.13a), use of the macro invokes a function whose generic parameters have the corresponding real type determined as follows:

In 7.25#5, replace:

For each unsuffixed function in **<math.h>** without a **c**-prefixed counterpart in **<complex.h>** (except **modf),**

with:

For each unsuffixed function in **<math.h>** without a **c**-prefixed counterpart in **<complex.h>** (except **modf**, **setpayload**, **setpayloadsig**, and **canonicalize**),

In 7.25#5, include in the list of type-generic macros: **roundeven**, **nextup**, **nextdown**, **fminmag**, **fmaxmag**, **llogb**, **fromfp**, **ufromfp**, **fromfpx**, **ufromfpx**, **totalorder**, and **totalordermag**.

After 7.25#6, add:

[6a] The functions that round result to a narrower type have type-generic macros whose names are obtained by omitting any ~~f or~~ **l** suffix from the function names. Thus, the macros are:

```
fadd                fmul                  ffma
dadd                dmul                  dfma
fsub                fdiv                  fsqrt
dsub                ddiv                  dsqrt
```

All arguments are generic. If any argument is not real, use of the macro results in undefined behavior. If any argument has type **long double**, or if the macro prefix is **d**, the function invoked has the name of the macro with an **l** suffix. Otherwise, the function invoked has the name of the macro (with no suffix).

[6b] A type-generic macro corresponding to a function indicated in the table in 7.6.1a is affected by constant rounding modes (7.6.2). Note that the type-generic macro definition in the example in 6.5.1.1 does not conform to this specification. A conforming macro could be implemented as follows:

```
#define cbrt(X) _Generic((X),                        \
                     long double: cbrtl(X),          \
                     default: _Roundwise_cbrt(X),    \
                     float: cbrtf(X)                 \
                     )
```

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

In 7.25#6a, attach a footnote to the wording:

> **`l`** suffix

where the footnote is:

> *) There are no functions with these macro names and the **`f`** suffix.

In 7.25#7, append to the table:

`fsub(f, ld)`	**`fsubl(f, ld)`**
`fdiv(d, n)`	**`fdiv(d, n)`**, the function
`dfma(f, d, ld)`	**`dfmal(f, d, ld)`**
`dadd(f, f)`	**`daddl(f, f)`**
`dsqrt(dc)`	undefined behavior

Bibliography

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

~~[2] ISO/IEC 9899:2011/Cor.1:2012, *Technical Corrigendum 1*~~

~~[3] ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-point arithmetic*~~

~~[4] 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*~~

~~[5~~[1] IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems, second edition*

~~[6] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*~~

~~[7~~[2] IEEE 754−1985, *IEEE Standard for Binary Floating-Point Arithmetic*

~~[8~~[3] IEEE 754-2008, *IEEE Standard for Floating-Point Arithmetic*

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

[5] ISO/IEC 9899:2011/Cor.1:2012, *Information technology — Programming languages — C / Technical Corrigendum 1*

